Computational cost estimates for parallel shared memory isogeometric multi-frontal solvers

M.Woźniak[(1)], K.Kuźnik[(1)], M. Paszyński[(1)], V. M. Calo[(2)], D. Pardo[(3)]

[(1)] *AGH University of Science and Technology, Krakow, Poland*
e-mail: macwozni@agh.edu.pl
e-mail: kmkuznik@gmail.com
e-mail: paszynsk@agh.edu.pl
[(2)] *King Abdullah University of Science and Technology, Thuwal, Saudi Arabia*
e-mail: victor.calo@kaust.edu.sa
[(3)] *University of the Basque Country, UPV/EHU and Ikerbasque, Bilbao, Spain*
e-mail: dzubiaur@gmail.com

Abstract

In this paper we present computational cost estimates for parallel shared memory isogeometric multi-frontal solver. The estimates show that the ideal isogeometric shared memory parallel direct solver scales as $O(p^2 \log(N/p))$ for one dimensional problems, $O(Np^2)$ for two dimensional problems, and $O(N^{4/3}p^2)$ for three dimensional problems, where N is the number of degrees of freedom, and p is the polynomial order of approximation. The computational costs of the shared memory parallel isogeometric direct solver are compared with those corresponding to the sequential isogeometric direct solver, being the latest equal to $O(Np^2)$ for the one dimensional case, $O(N^{1.5}p^3)$ for the two dimensional case, and $O(N^2p^3)$ for the three dimensional case. The shared memory version significantly reduces both the scalability in terms of N and p. Theoretical estimates are compared with numerical experiments performed with linear, quadratic, cubic, quartic, and quintic B-splines, in one and two spatial dimensions.

Keywords: isogeometric finite element method, multi-frontal direct solver, computational cost, NVIDIA CUDA GPU

1. Introduction

Classical higher order finite element methods (FEM) [17, 18] maintain only C^0-continuity at the element interfaces, while isogeometric analysis (IGA) utilizes B-splines as basis functions, and thus, it delivers C^k global continuity [14]. The higher continuity obtained across elements allows IGA to attain optimal convergence rates for any polynomial order, while using fewer degrees of freedom [3, 1]. Nevertheless, this reduced count in the number of degrees of freedom may not immediately correlate with a computational cost reduction, since solution time per degree of freedom augments as the continuity is increased [10, 13]. In spite of the increased cost of higher-continuous spaces, they have proven very popular and useful. For example, higher-continuous spaces have allowed the solution of higher-order partial differential equations with elegance [7, 28, 29, 51, 16, 15] as well as several non-linear problems of engineering interest [31, 6, 30, 5, 20, 11, 9, 4]. Thus, efficient multi-frontal solvers for higher-continuous spaces are important.

The multi-frontal solver is one of the state-of-the art algorithm for solving linear systems of equations [22, 26]. It is a generalization of the frontal solver algorithm proposed in [33, 21]. The multi-frontal algorithm constructs an assembly tree based on the analysis of the connectivity data or the geometry of the computational mesh. Finite elements are joint into pairs and fully assembled unknowns are eliminated within frontal matrices associated to multiple branches of the tree. The process is repeated until the root of the assembly tree is reached. Finally, the common interface problem is solved and partial backward substitutions are recursively called on the assembly tree.

There exist parallel versions of the multi-frontal direct solver algorithm targeting distributed-memory, shared-memory, or hybrid architectures. The partition of data for distributed memory architecture may concern the re-distribution of the computational mesh into sub-domains with overlapping or non-overlapping elements [47, 48], the redistribution of the global matrix [27], or the redistribution of the elimination tree [44, 45, 46, 50].

There also exist some versions of the multi-frontal solver algorithm for shared memory machines. These algorithms store the entire matrix in the shared memory and perform matrix operations concurrently [23, 24, 25]. The matrix is partitioned into blocks with BLAS operations performed concurrently over each block.

One limitation of the IGA is the fact that direct solvers work slower, due

to *inconvenient* sparsity pattern of the resulting matrix [13]. In particular, the sequential IGA direct solvers delivers $O((N/p)p^2)$ computational cost for one dimensional problems (1D), $O(N^{1.5}p^3)$ for two dimensional problems (2D), and $O(N^2p^3)$ for three dimensional problems (3D) [13]. In those estimates, we assumed uniform p-order B-spline basis functions over the entire mesh delivering C^{p-1} global regularity of the solution. The direct solver algorithm for IGA delivering C^{p-1} global regularity with p-order B-splines is p^3 times slower than the direct solver with p-order polynomial with C^0 global regularity, for 2D and 3D problems, and p^2 times slower for 1D problems. Thus, for $p = 3$, the C^2 global continuity problem is 27 times more expensive to be solved than the C^0 counterpart in 2D and 3D, which implies that we may need to wait days instead of hours to obtain the solution.

The main contribution of this paper is to show theoretically and experimentally that the IGA sequential solver limitations can be overcome to some extent by utilizing shared memory GPU implementations. In this paper, we present the derivation of the computational cost for an ideal shared memory parallel direct solver, where we assume zero communication costs, and uniform grids in terms of h (element size) and p (polynomial order of approximation). The computational cost estimations imply that the ideal IGA shared memory parallel direct solver delivers $O(p^2 \log(N/p))$ for 1D problems, $O(Np^2)$ for 2D problems and $O(N^{4/3}p^2)$ for 3D problems.

With these results, not only the N dependence is significantly improved by using the shared memory version, but also the p-dependence, which makes IGA more competitive with respect to traditional C^0-FEM when using shared memory machines. In particular, for $p = 3$, the C^2 global continuity problem becomes *only* 9 times more expensive to be solved than the C^0 counterpart in 2D and 3D, as opposed to 27 in the sequential version.

The first part of the paper presents estimates of computational costs and scalability for any shared memory implementation of multifrontal IGA solvers. The second part of the paper describes an implementation of the multi-frontal algorithm for graphics processors. The code is presented, its performance tested for several GPUs and the results are used for verifying the estimates derived in the first part. Among other things, this combination between theoretical and numerical results enables to show that the leading term (scalability) of our solver is not destroyed by implementation/architectural issues such as memory access. Notice that we are not assuming that the cost of memory access is zero, but rather that it is of lower (or equal) order that the number of floating point operations, which ultimately determines

the scalability of the solver, as shown by the numerical results.

In some recent papers [34, 35], the influence on execution time of the implementation factors, such as processor occupancy, thread synchronization, organization of memory accesses etc. is analyzed. However, in such a complex algorithm as the one presented here, a detailed analysis of memory access, synchronization, organization of memory accesses, etc. is extremely difficult to perform, and it is out of the scope of this paper. Rather, numerical results show that the theoretical scalability is indeed achievable in practice, thus, showing that other factors (including memory access) will not destroy the predicted scalability of the solver.

The theoretical computational cost estimates are compared with our parallel shared memory implementation. We target our solver NVIDIA's GPU architecture, which is a complex shared memory architecture. We tested our implementations on a GeForce GTX 560 Ti device with 8 multiprocessors, each one equipped with 48 cores. as well as on NVIDIA Tesla C2070 device, which has 14 multiprocessors with 32 CUDA cores per multiprocessor. We also performed tests for the 2D solver on a GeForce GTX 780 graphic card equipped with 3 gigabytes of memory and 2304 cores.

For the description of a B-spline based multi-frontal solver algorithm for 1D problems, we refer to [37]. The 1D IGA solver is similar to those used in finite difference methods [41]. For a detailed description of the B-spline based multi-frontal solver for 2D problems, we refer to [36].

The structure of the paper is the following. We start with the definition of our model problem in Section 2. Section 3 describes the basics of the isogeometric analysis using B-splines. Section 4 introduces the multi-frontal solvers algorithm for IGA. Section 5 presents the computational cost and memory usage estimates for the 1D, 2D, and 3D multi-frontal shared-memory parallel, direct solver algorithm. In Section 6, we describe the technical details related to the implementation of the 1D and 2D multi-frontal solver. Section 7 presents the numerical results for 1D and 2D models as well as a short discussion on the limitations of the 3D implementation. Finally, Section 8 depicts the main conclusions of this work and possible lines of research for the future.

2. Model problem

In this section, we present our model problem. We focus on the 2D conductive media equation

$$\nabla \cdot \sigma \nabla u = \nabla \cdot \mathbf{J}^{imp}, \tag{1}$$

where σ is the conductivity of the media, u is the electric potential, and $\mathbf{J}^{imp}$ is the impressed electric current (the source). The above partial differential equation (PDE) is imposed on a computational domain $\Omega = [0,1]^d$, where d is the spatial dimension. We impose homogeneous Dirichlet boundary conditions

$$u = 0 \text{ on } \Gamma_D, \tag{2}$$

where $\Gamma_D = \partial\Omega$.

The weak variational formulation is obtained by taking the L^2-scalar product with functions $v \in H^1_{\Gamma_D}(\Omega) = \{v \in H^1(\Omega) : v|_{\Gamma_D} = 0\}$, and integrating by parts to obtain:

$$\text{Find } u \in V = H^1_{\Gamma_D}(\Omega) \text{ such that} \tag{3}$$

$$b(v,u) = l(v), \forall v \in V, \tag{4}$$

where

$$b(v,u) = \int_\Omega \sigma \nabla v \cdot \nabla u dx, \text{ and} \tag{5}$$

$$l(v) = \int_\Omega v \cdot \mathbf{J}^{imp} dx \tag{6}$$

The computational cost of the solver is independent of the considered PDE as long as it is given by a single scalar equation. A different equation would modify the values in the frontal matrices, but the location of the zeros would remain unaffected. When switching to a system of equations, the element matrices become blocks related to the number of equations, and this factor N_{eq} will influence the scalability of the solver like the p factor. In other words, defining $\tilde{p} := pN_{eq}$, all our estimates remain valid after replacing p by $\tilde{p}$.

The computational cost of the solver is also independent of geometrical variations. This is because our B-spline basis functions can model complex

geometries on our patch of elements. The geometry may affect the value of the Jacobian in the integrals, but not the location of the nonzero entries. Our study however is limited to a single patch of elements. In a case of multiple patches, the solver should return the Schur complements with respect to the boundary of each patch being used, and the Schur complements must be processed on the higher level by some external solver.

3. Isogeometric analysis (IGA) using B-splines

Contrary to the classical *hp* adaptive FEM, e.g. as in Leszek Demkowicz book [17, 18], which provides C^0 global continuity with linear basis functions over element vertexes and bubble functions over element edges and interiors, the IGA provides C^{p-1} global continuity, and it utilizes B-spline functions as basis functions.

The classical IGA-FEM [14] considers the patches of elements, defined as tensor products of knot vector, with uniform polynomial order of approximation, resulting in up to C^{p-1} global continuity. Namely, for the 1D IGA-FEM case, the computational mesh is a uniform patch of N_x elements with uniform order of approximation p, with basis functions defined as B-splines, which spread over $p+1$ elements. For the 2D IGA-FEM case the computational mesh is a uniform patch of $N_x N_y$ elements, with uniform order of approximation p, with basis functions defined as tensor products of B-splines, which spread over $(p+1)^2$ elements. For the 3D IGA-FEM case the computational mesh is a uniform patch of $N_x N_y N_z$ elements with uniform order of approximation p, with basis functions defined as tensor products of B-splines, which spreads over $(p+1)^3$ elements.

In this work, we refer to this classical IGA-FEM setup and we analyze the uniform p refinement over a patch of elements. The uniform p refinements themselves are important [2] for the case of IGA-FEM, since they increase the global regularity of the solution up to C^{p-1} .

In the 1D case, we approximate the solution u and the test function v with B-spline basis functions

$$u(x) \approx \sum_i N_{i,p}(x)\, a_i \qquad v \in \{N_{j,p}\}_j \tag{7}$$

where $i, j = 1, ..., N_x + p$ and

$$N_{i,0}(\xi) = I_{[\xi_i, \xi_{i+1}]} \tag{8}$$

$$N_{i,p}(\xi) = \frac{\xi - \xi_i}{x_{i+p} - \xi_i} N_{i,p-1}(\xi) + \frac{\xi_{i+p+1} - \xi}{x_{i+p+1} - \xi_{i+1}} N_{i+1,p-1}(\xi), \tag{9}$$

with p being the polynomial order of the B-spline basis functions and $I_{[\xi_i,\xi_{i+1}]}$ the characteristic function over the interval $[\xi_i, \xi_{i+1}]$, that is, $I_{[\xi_i,\xi_{i+1}]}(x) = 1$ if $x \in [\xi_i, \xi_{i+1}]$ and 0 otherwise. Examples of linear and quadratic B-splines are given in Figure 1.

Substituting these definitions in the weak form allows us to obtain the discrete weak formulation, which can be stated as follows,

$$\sum_i b\left(N_{j,p}(x), N_{i,p}(x)\right) a_i = l\left(N_{j,p}(x)\right), \forall j \tag{10}$$

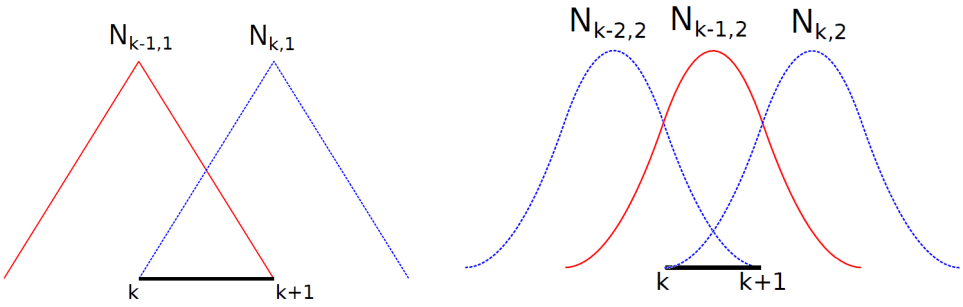

Figure 1: Linear (left panel) and quadratic (right panel) B-splines basis functions with support over element $[k, k+1]$.

For each row in (10), the integral is restricted to the support of $N_{j,p}$. This support is broken into elements, where $N_{i,p}$ and $N_{j,p}$ are simple polynomials. The integral $b\left(N_{j,p}, N_{i,p}\right)$ restricted to an element e is denoted as the element stiffness matrix, while the restriction of $l\left(N_{j,p}\right)$ to the element is denoted as the element force vector.

In two (and higher) dimensions, basis functions are tensor products of one dimensional B-splines

$$u(\xi, \eta) \approx \sum_{i,j} B_{i,j;p}(\xi, \eta) U_{i,j} = \sum_{i,j} N_{i,p}(\xi) N_{j,p}(\eta) U_{i,j} \tag{11}$$

We end up with the following discrete form of the weak equation

$$\text{Find } u \in V = \sum_{i,j} B_{i,j;p} U_{i,j} = \sum_{i,j} N_{i,p} N_{j,p} U_{i,j} \text{ such that} \tag{12}$$

$$\sum_{i,j} b\left(B_{k,l;p}, B_{i,j;p}\right) = l\left(B_{k,l;p}\right) \forall k, l \tag{13}$$

The number of B-splines over a single finite element $E_{k,l} = [\xi_k, \xi_{k+1}] \times [\eta_l, \eta_{l+1}]$ is equal to $(p+1)^2$. The element local matrix has the following form:

$$\{B_{m,n;p}(x_1, x_2)\}_{m=k-p,\ldots,k;n=l-p,\ldots,l} = \{N_{m;p}(x_1), N_{n;p}(x_2)\}_{m=k-p,\ldots,k;n=l-p,\ldots,l} \tag{14}$$

The element matrix contributions are computed by using Gaussian quadrature rules with weights w_r and points $\tilde{x^r} = (x_1^r, x_2^r)$

$$b(B_{i,j;p}, B_{k,l;p}) = \int_\Omega \nabla B_{i,j;p} \cdot \nabla B_{k,l;p} dx = \tag{15}$$

$$\sum_r w_r \sum_{m=1,2} \frac{\partial}{\partial x_m} \left(N_{i;p}(x_1) N_{j;p}(x_2)\right)(x_1^r, x_2^r) \frac{\partial}{\partial x_m} \left(N_{k;p}(x_1) N_{l;p}(x_2)\right)(x_1^r, x_2^r) \tag{16}$$

$$l(B_{k,l;p}) = -\int_\Omega \frac{\partial B_{k,l;p}}{\partial x_2} dx = \sum_r w_r \frac{\partial}{\partial x_2} \left(N_{k;p}(x_1) N_{l;p}(x_2)\right)(x_1^r, x_2^r) \tag{17}$$

In order to compute the element local matrix contributions for B-splines of order p, it is necessary to compute values of 1D B-splines $N_{k;p}$ and their derivatives at Gaussian integration points x_i^r.

Remark 1 *Gaussian quadrature makes IGA inefficient when compared with other weak-form-based methods, such as, hp-finite elements [17, 18] and spectral methods [8]. That is, on average, IGA requires $p-1$ evaluations per function, rather than one per function as alternative methods need. More efficient rules have been proposed in [32]. The objective is to perform only one function evaluation per degree of freedom in order to make IGA as efficient as alternative methods with respect to numerical integration. The quadrature rules are computed on the fly by solving a system of equations for a given knot vector in each parametric direction. Given the numerical difficulties of obtaining the collocation points and weights with the necessary accuracy when inverting the resulting system of equations, and since this is not the focus of this paper, we have chosen to use conventional Gauss quadrature instead.*

4. Direct Solver Algorithm

4.1. The Schur Complement

We first analyze the FLOPS and memory cost of performing the Schur complement operation, which is the main building block for construction of a multi-frontal solver.

Let matrix B be decomposed as:

$$B = \begin{bmatrix} \mathrm{C} & \mathrm{D} \\ \mathrm{E} & \mathrm{F} \end{bmatrix}, \tag{18}$$

where square matrices C and F have dimensions q and r, respectively.

The Schur complement method consists of performing partial LU factorization of the square submatrix C to obtain:

$$B = \begin{bmatrix} \mathrm{I} & 0 \\ \mathrm{EC}^{-1} & \mathrm{I} \end{bmatrix} \begin{bmatrix} \mathrm{C} & 0 \\ 0 & \mathrm{F\text{-}EC}^{-1}\mathrm{D} \end{bmatrix} \cdot \begin{bmatrix} \mathrm{I} & \mathrm{C}^{-1}\mathrm{D} \\ 0 & \mathrm{I} \end{bmatrix}. \tag{19}$$

Term F-EC^{-1}D is the so-called Schur complement. For the sequential version, we have (see [13]):

$$\begin{aligned} &\text{FLOPS} = \mathcal{O}(q^3 + q^2 r + q r^2) = \mathcal{O}(q^3 + q r^2) \\ &\text{Memory} = \mathcal{O}(q^2 + qr) \end{aligned} \tag{20}$$

Note that better estimates in terms of FLOPS can be obtained using fast matrix-matrix multiplication algorithms, as described in [49] and references therein. However, with rare exceptions, such approaches often assume extremely large values of q, in some cases, thousands of millions, which limit their practical use in general applications. For the sake of simplicity, we avoid their use here.

In the above memory estimate, we are only concerned with the space required to store factors L and U, since it is well-known that the cost of storing original matrix B is always smaller or equal than the memory required to store L and U.

In particular, we have not included the memory required to store the Schur complement, since on the next step of the LU factorization, this Schur complement is further factorized in terms of other Schur complements until it has dimension zero, and therefore, occupies no memory.

For the shared-memory parallel version, the amount of memory remains constant in the best case scenario, independently of the number of processors N_{proc}. In terms of FLOPS, from equation (20) we obtain that the maximum number of FLOPS per core (MFPC) is given by:

$$\text{MFPC} = \mathcal{O}\left(\frac{q^3}{\min\{N_{proc}, q\}} + \frac{q r^2}{\min\{N_{proc}, r\}}\right). \tag{21}$$

Notice that for a sufficiently large number of processors N_{proc}, the maximum number of FLOPS per core reduces to simply $\mathcal{O}(q^2 + qr)$, which coincides with the needed amount of memory.

4.2. The multi-frontal solver

We divide our computational domain in N_p patches. For IGA, each patch consists of a set of p consecutive elements in each dimension. For simplicity, we assume that the number of patches of our computational domain is $(2^s)^d$, where s is an integer, and d is the spatial dimension of the problem. Notice that even when this assumption is not verified, the final result still holds true. The initial patches contain $(p+1)^d$ elements.

The idea of the multi-frontal solver is to eliminate interior unknowns of each patch, then join each 2^d patches into one to produce $(2^{s-1})^d$ new patches, eliminate interior unknowns of each new patch, and continue with the iterative procedure until the last 2^d patches are joint into one. The iterative algorithm can be expressed as follows:

- `for` $i = 0, s-1$:
- $N_p = N_p(i) = (2^{s-i})^d$
- `If` $i = 0$, `define` $N_p(0)$ `patches. Otherwise, join old` $N_p(i-1)$ `patches to define` $N_p(i)$ `new patches.`
- `Eliminate interior unknowns of each patch.`
- `end loop.`
- `merge the last` 2^d `patches and solve the top problem.`

Let us illustrate the above algorithm with a 2D example for second order B-splines. Let us assume we have $d = 2$ and $s = 3$, in other words, we have $(2^s)^d = (2^3)^2 = 8^2 = 64$ patches over the mesh.

- In the first step, we have $N_p(0) = (2^{s-0})^d = (2^3)^2 = 64$ patches. Each patch contains $(p+1)^2$ elements. The single element is illustrated on panel (a) of Figure 2, while the initial patch with $(p+1)^2 = 3^2 = 9$ elements is illustrated on panel (b) of Figure 2. In this first step, we can eliminate one interior B-spline, denoted with red color on panel (b) of Figure 2.

- In the second step, we have $N_p(1) = (2^{s-1})^d = (2^2)^2 = 8$ patches. These patches are obtained by merging $2^d = 2^2 = 4$ patches from the previous step. An exemplary patch from this step is presented on panel (c) of Figure 2. We can eliminate twelve fully assembled B-splines denoted on panel (c) with red and green colors.

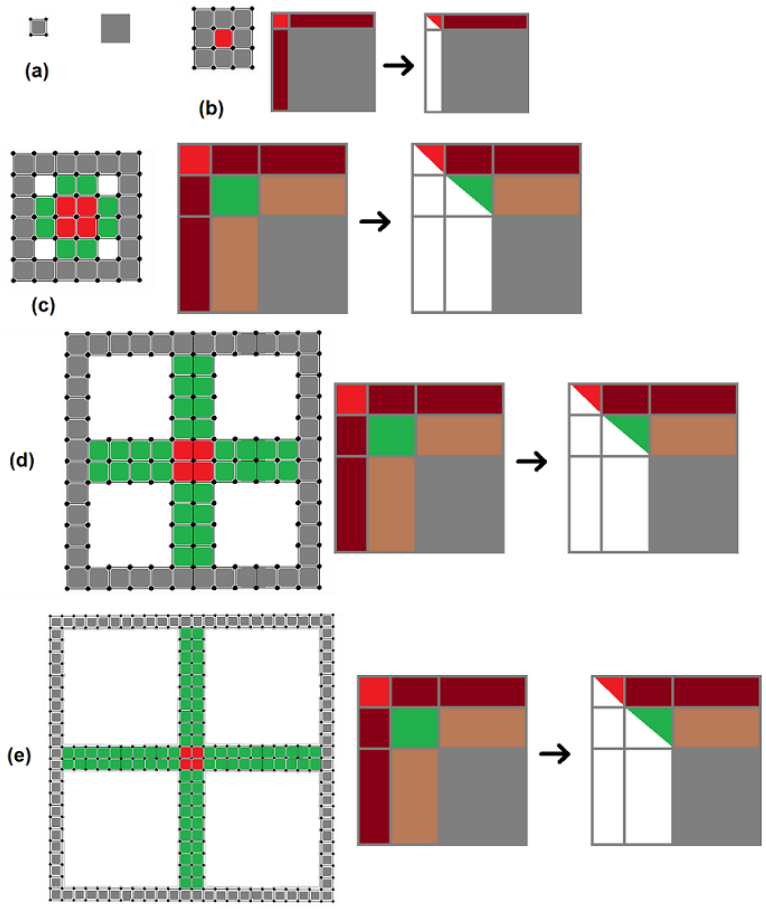

Figure 2: Summary of the factorization step for the 2D solver.

- In the third step, we have $N_p(2) = (2^{s-2})^d = (2^1)^2 = 4$ patches. These patches are obtained by merging $2^d = 2^2 = 4$ patches from the previous step. An exemplary patch from this step is presented on panel (d) of Figure 2. We can eliminate 36 fully assembled B-splines denoted on panel (d) with red and green colors.

- In the last step, we have a single patch obtained by merging $2^d = 2^2 = 4$ patches from the previous step. An exemplary patch from this step is presented on panel (e) of Figure 2. We can eliminate all B-splines at this point, since all the B-splines are fully assembled.

Notice that all patches can be processed concurrently at any given step of

the factorization. Also, if we have enough cores, the row subtractions during the partial eliminations can be performed concurrently, since the matrices are assumed to be stored in shared memory.

5. Theoretical Complexity Estimates for Direct Solvers

In this section, we derive estimates for the number of FLOPS and memory required to solve a system of linear equations using a direct multi-frontal solver on a shared memory parallel machine.

For the sequential version, the FLOPS and memory required by the above algorithm can be expressed as:

$$\sum_{i=0}^{s-1} N_p(i) \cdot S(i), \tag{22}$$

where $S(i)$ is the cost (either FLOPS or memory) of performing each Schur complement at the i-th step. Following the notation of the previous subsection on the Schur complement, we define $q = q(i)$ as the number of interior unknowns of each path at the i-th step, and $r = r(i)$ as the number of interacting unknowns at the i-th step.

For the shared-memory version, we distribute at the i-th step all $N_p(i)$ patches over N_{proc} processors, and if $N_{proc} > N_p(i)$, the cost $S(i)$ over each patch will be further subdivided among the available processors per patch. Thus, the total number of FLOPS becomes:

$$\sum_{i=0}^{s-1} \frac{N_p(i)}{\min\{N_p(i), N_{proc}\}} \cdot \mathcal{O}(\frac{q(i)^3}{\min\{\tilde{N}_{proc}(i), q(i)\}} + \frac{q(i)r(i)^2}{\min\{\tilde{N}_{proc}(i), r(i)\}}), \tag{23}$$

where $\tilde{N}_{proc}(i) = \max\{1, \frac{N_{proc}}{N_p(i)}\}$. Notice that for a sufficiently large number of processors N_{proc}, the number of FLOPS reduces to:

$$\sum_{i=0}^{s-1} \mathcal{O}(q(i)^2 + q(i)r(i)). \tag{24}$$

For this case, taking into account the number of interior and interacting unknowns at each step of the multi-frontal solver (Table 1), and combining them with the cost of Schur complement operations given by Equations (20) and (21) and the FLOPS and memory estimates (22) and (23), we conclude:

	q(0)	r(0)	q(i) , $i \neq 0$	r(i) , $i \neq 0$
hp-FEM	$\mathcal{O}(p^d)$	$\mathcal{O}(p^{d-1})$	$\mathcal{O}(2^{(d-1)i}p^{d-1})$	$\mathcal{O}(2^{(d-1)i}p^{d-1})$
IGA	$\mathcal{O}(1)$	$\mathcal{O}(p^d)$	$\mathcal{O}(2^{(d-1)i}p^d)$	$\mathcal{O}(2^{(d-1)i}p^d)$

Table 1: Number of interior and interacting unknowns at each step of the multi-frontal solver.

- 1D IGA:

$$\begin{aligned} \text{FLOPS} = &\quad 2^s p^2 + \sum_{i=1}^{s-1} 2^{s-i} p^3 = \mathcal{O}(2^s p^3) = \mathcal{O}(N_p p^3) = \mathcal{O}(Np^2), \\ \text{MFPC} = &\quad p + \sum_{i=1}^{s-1} p^2 = \mathcal{O}(p^2 \log N_p) = \mathcal{O}(p^2 \log(N/p)), \\ \text{Memory} = &\quad 2^s p + \sum_{i=1}^{s-1} 2^{s-i} p^2 = \mathcal{O}(2^s p^2) = \mathcal{O}(N_p p^2) = \mathcal{O}(Np). \end{aligned} \tag{25}$$

- 2D IGA:

$$\begin{aligned} \text{FLOPS} = &\quad 2^{2s} p^4 + \sum_{i=1}^{s-1} 2^{2(s-i)} 2^{3i} p^6 = \mathcal{O}(2^{2s} p^4 + 2^{3s} p^6) = \\ &\quad \mathcal{O}(N_p^3 p^6) = \mathcal{O}(N^{1.5} p^3) \\ \text{MFPC} = &\quad p^2 + \sum_{i=1}^{s-1} 2^{2i} p^4 = \mathcal{O}(2^{2s} p^4) = \mathcal{O}(N_p^2 p^4) = \mathcal{O}(Np^2) \\ \text{Memory} = &\quad 2^{2s} p^2 + \sum_{i=1}^{s-1} 2^{2(s-i)} 2^{2i} p^4 = \mathcal{O}(2^{2s} p^2 + s^2 2^{2s} p^4) = \\ &\quad \mathcal{O}(N_p^2 p^4 \log(N_p^2/p^2)) = \mathcal{O}(p^2 N \log(N/p)) \end{aligned} \tag{26}$$

- 3D IGA:

$$\begin{aligned} \text{FLOPS} = &\quad 2^{3s} p^6 + \sum_{i=1}^{s-1} 2^{3(s-i)} 2^{6i} p^9 = \mathcal{O}(2^{3s} p^6 + 2^{6s} p^9) = \\ &\quad \mathcal{O}(N_p^3 p^6 + N_p^6 p^9) = \mathcal{O}(N^2 p^3) \\ \text{MFPC} = &\quad p^4 + \sum_{i=1}^{s-1} 2^{4i} p^6 = \mathcal{O}(2^{4s} p^6) = \mathcal{O}(N_p^4 p^6) = \mathcal{O}(N^{4/3} p^2) \\ \text{Memory} = &\quad 2^{3s} p^4 + \sum_{i=1}^{s-1} 2^{3(s-i)} 2^{4i} p^6 = \mathcal{O}(2^{3s} p^4 + 2^{4s} p^6) = \\ &\quad \mathcal{O}(N_p^4 p^6) = \mathcal{O}(p^2 N^{4/3}) \end{aligned} \tag{27}$$

Thus, for a sufficiently large N_{procs}, we conclude the following:

1. Results for 1D, 2D, and 3D are essentially different.
2. For 2D and 3D, the number of FLOPS of the IGA method grows as p^3 for the sequential version and as p^2 for the shared memory parallel version. Thus, an IGA grid-adaptive algorithm *should not* be based exclusively on the maximum decrease of the error per added unknown. It should also incorporate a special treatment of the cost of each added unknown depending upon the type of refinement.

3. For 2D and 3D, the amount of memory required by the IGA method also grows as p^2.
4. For the IGA discretization, the scalability of the shared memory parallel version degenerates as $O(\log(N/p))$ for 1D, as $O(N^{0.5}/p)$ for 2D, and as $O(N^{0.33}/p)$ for 3D. We refer here to the degeneration of scalability with respect to the perfect strong scalability. Curiously, the largest scalability degeneration as the problem size increases is observed in 2D. In other words, the 2D case is the one that parallelizes "the worst".

Conversely, if $N_{proc} \leq \min\{N_p(i) \cdot q(i), N_p(i) \cdot r(i)\} \quad \forall i$, then we see from Equation (23) that we recover a perfect scalability, that is, the maximum number of FLOPS per core is of the order of the total number of FLOPS in the sequential version divided by N_{proc}. When the number of processors is in between the small and large limits studied above, the resulting scalability is also in between the perfect one obtained for a small number of processors and the one obtained in Equations (25)-(27) for a sufficiently large number of processors. The specific formula for any number of processors can be derived from Equation 23 and Table 1, although it becomes hard to interpret due to the multiple minima and maxima that appear in the formulas.

6. Algorithmic implementation

We employ CUDA (Compute Unified Device Architecture) for our implementation designed for NVIDIA GPUs.

To quickly outline modern GPU architectures, they consists of several multiprocessors, each containing many cores. Moreover, there are 4 kinds of memory: global, shared, constant, and texture. For this article we are interested only in the two former types of memory. We could use constant and texture to speed up computations, although the algorithm then would be tied to CUDA forever. Global memory can be accessed from every multiprocessor, but its latency is considerably high, while the shared memory can be accessed by all threads running on one multiprocessor. This memory is really small, but comparing to global memory, it has a much lower latency and a much higher throughput. Our implementation tries to use global memory as efficiently as possible (avoiding scattered access) and makes use of shared memory to speed up computations.

In this section, we summarize the one and two dimensional solver implementations.

1. In the first step, we generate the element local matrices.
    - In the 1D case, each thread is responsible for calculating values of B-splines and their derivatives at quadrature points over one element. We start with linear B-splines and calculate higher orders splines hierarchically. Using those values, we generate local frontal matrices. Here, we run one thread per matrix entry.
    - In the 2D case, local matrices are generated by evaluating functions $b(B_{ij}, B_{kl})$ over each element. We run a grid of $n \times n$ blocks with $(p+1)^2$ by $(p+1)^2$ threads each. The right hand side $l(B_{kl})$ is computed over a grid of $n \times n$ blocks with $(p+1)^2$ threads each. All operations are implemented as weighted summation over Gaussian quadrature points.
2. The second step consists of merging the frontal matrices recursively, and eliminating the fully assembled rows
    - In the 1D case, frontal matrices are small enough so we can fit several of them into shared memory. After the first piece of data is loaded into shared memory, we perform merging and elimination and store the corresponding results into the global memory.
    - In the 2D case, in the initial merging step we join $(p+1)^2$ matrices to fully assemble just one B-spline. We run a 2D grid of blocks, where each block is responsible for merging $(p+1)^2$ matrices into one. In the following steps, we merge the resulting Schur complement matrices and eliminate fully assembled rows, level by level, until we reach the root of the elimination tree. These steps are the most time consuming parts of the solver algorithm. Merging is divided into horizontal and vertical merges to limit the number of rows eliminated in one step and to make better use of fast shared memory. At each level of the elimination tree, we run as many blocks as merged matrices we receive. The number of threads in a block depends on the level of the elimination tree. At first, there are as many threads as the length of a row in the contributing matrix. Then, the number of threads is equal to the number of rows eliminated in the current step. Finally, it becomes equal to the number of rows that are not eliminated. We merge matrices until we reach the top of the elimination tree, where we are left with dense problem.

3. In the next step, we construct the top dense fully assembled problem and solve it.
    - In the 1D case, the final merge, full elimination, and first backward substitution is executed by a single thread.
    - In the 2D case, the final dense problem is solved by calling the dense solver library MAGMA [39].
4. Backward substitution.
    - In the 1D case, we use values stored in global memory. Recursively, we travel through the elimination tree and use one thread per variable, which is calculated on the current level of backward substitution.
    - In the 2D case, backward substitutions are executed over the same elimination tree, but coming from the top of the tree down to the leaves. For each block, we run as many threads as there were eliminated rows.
5. Generation of the solution values.
    - In the 1D case, we multiply the coefficients by our basis functions to obtain the final solution. One thread per basis function is utilized.
    - In the 2D case, we also have the coefficients for every basis function. With those coefficients, we are able to calculate values of the solution at any point. Actually, we run a grid of $n \times n$ blocks with number of threads equal to the number of quadrature points in one element.

7. Numerical Results and Discussion

This section compares the theoretical estimates with numerical experiments. The 1D experiments were performed on GeForce GTX 560 Ti graphic card with 8 multiprocessors, each one equipped with 48 cores. The total number of cores is equal to 384. The global memory on this graphics card was 1024MB. The 2D experiments were performed on two graphic cards. The first one was NVIDIA Tesla C2070 device, which has 14 multiprocessors with 32 CUDA cores per multiprocessor. The second one was GeForce GTX

780 , with 2304 CUDA cores, 899 MHz. The total amount of memory was 3 gigabytes.

The numerical measurements presented in this paper include the integration, the factorization, and the backward substitution times. This applies both to the GPU and CPU measurements. We show these results because for the considered cases, the integration and backward substitution execution times are of the same or smaller order than the factorization time, thus, they do not influence the trends (scalability) of the computational cost, they only affect the constants. This shows that the total cost is dominated by the factorization cost. Thus: (a) results scale according as those corresponding to the LU factorization only, and (b) we show that the cost of the LU factorization is dominating the entire computation.

The measurements for the numerical results have been obtained by using the following tools: (a) execution time for MUMPS has been obtained by analyzing log file from PetIGA [12] calls, (b) number of GFLOPS for GPU has been measured using the profiler, (c) memory usage for GPU has been measured using the GPU status checker `nvidia-smi` tool, and (d) execution time for GPU solver has been measured by calling the boost library from inside of the application from the CPU. The execution times include memory initialization and input/output data transfers. We have utilized different kernels for different steps of the algorithm.

7.1. 1D case

Numerical experiments were performed with linear, quadratic, cubic, quartic, and quintic B-splines. We report the time spent on execution of the multi-frontal solver, as well as the memory usage.

In Figure 3 (left panel), we observe straight lines in the natural vs. log scale, as predicted by the theory. Furthermore, the slopes are given by p^2, as expected. In terms of memory (right panel of Figure 3), we obtain a straight line in the linear vs. linear scale. Additionally, we also observe that all curves (almost) coincide, since the displayed memory has been divided by p, following the scaling dictated by the theoretical estimates.

We have also compared the numerical results for the 1D GPU solver with numerical results for the 1D sequential solver, namely with the MUMPS [40] state-of-the-art solver, executed in sequential mode on CPU. The results of the comparison are displayed in Figures 4-7 for linear, quadratic, cubic, and quartic B-splines. The horizontal axis uses the logarithmic scale, while the vertical axis uses the linear scale. For such a case, the GPU solver is expected

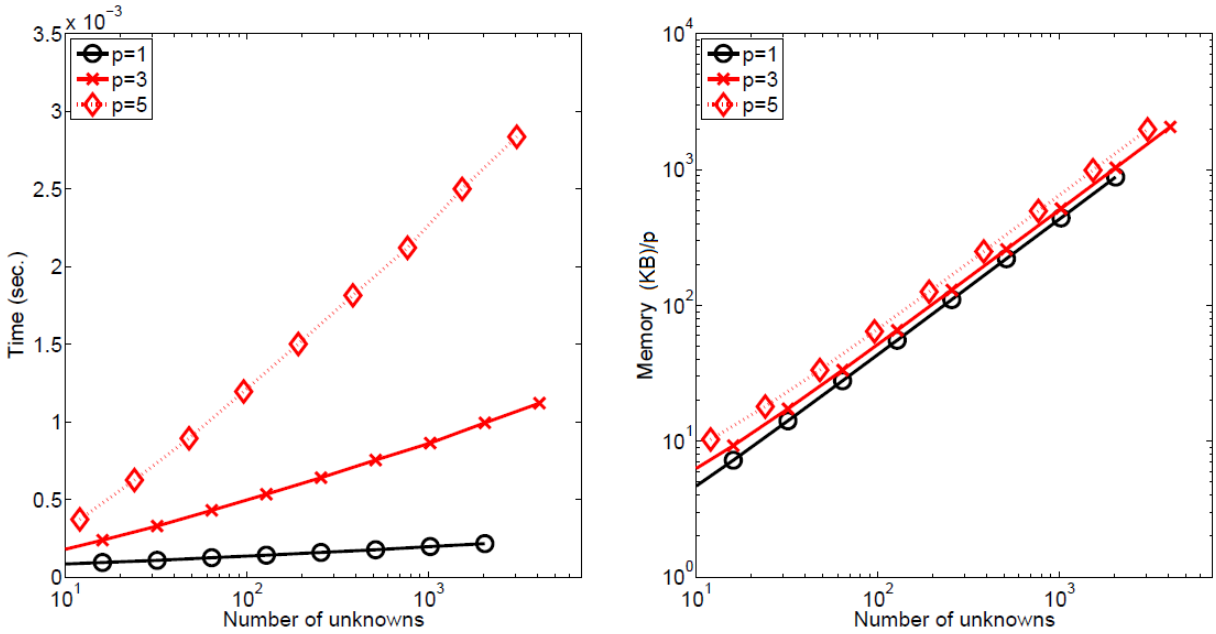

Figure 3: Execution time (left) and memory usage (right) measured on GPU for linear, cubic, and quintic 1D B-splines solver.

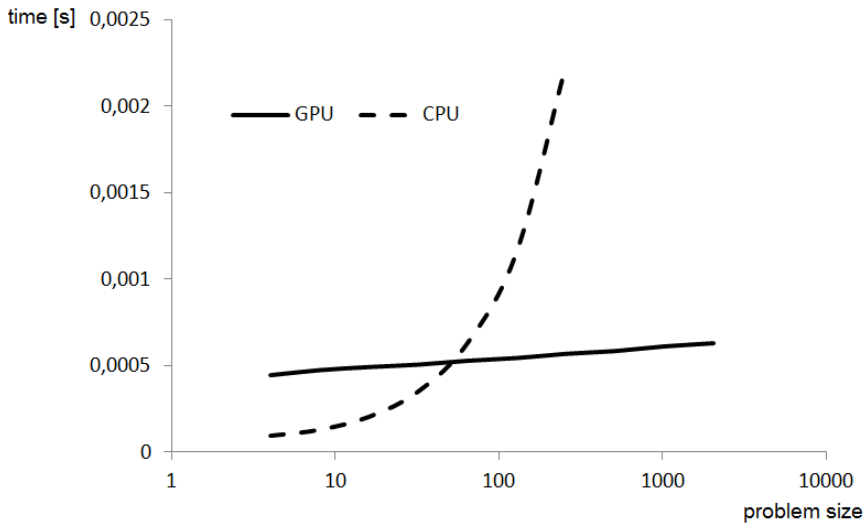

Figure 4: One dimensional solver using linear B-splines. Comparison of the GPU cost vs the CPU one.

to deliver $O(p^2 log(N/p))$ computational cost. For a fixed p, this simplifies into a $O(logN)$ behavior, resulting in a straight line on the log-linear scale. Notice that the sequential solver should deliver $O(Np^2)$ computational cost, which for fixed p results in $O(N)$ behavior. Thus, we observe a rapidly growing curve in the log-linear scale.

In the 1D case, the frontal matrices processed by the solver, which are assigned to different nodes of the elimination tree and located at different levels of the tree, are of the same size. In other words, the solver algorithm processes the elimination tree level by level, from the leaves up to the root. Processing of levels is interchanged with global synchronization barriers. If we assume a constant problem size, increasing the number of processors will just increase the number of nodes at a particular level of the tree that can be processed fully in parallel, see Figure 8.

If there are more processors than nodes at a given level of the elimination tree, the tree level can be processed fully in parallel. If the number of processors is less than the number of nodes, the tree level will be processed part by part. This is illustrated in Figure 9, where we measure the scalability of the 1D solver for 2048 elements, as we increase the number of working cores. It should be emphasized that this plot is based on the conceptual analysis

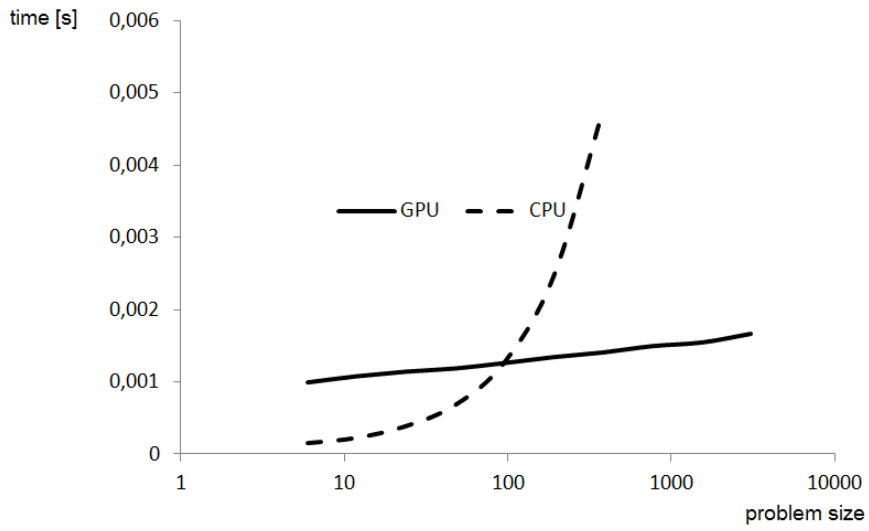

Figure 5: One dimensional solver using quadratic B-splines. Comparison of the GPU cost vs the CPU one.

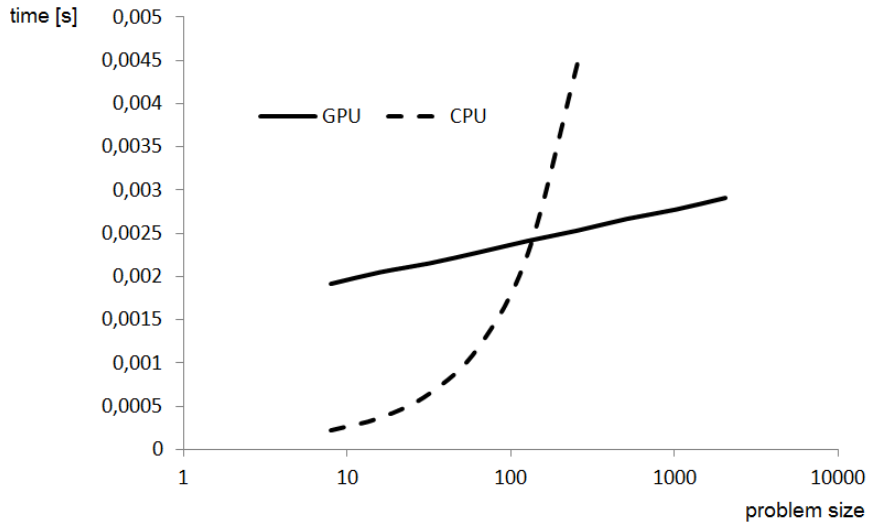

Figure 6: One dimensional solver using cubic B-splines. Comparison of the GPU cost vs the CPU one.

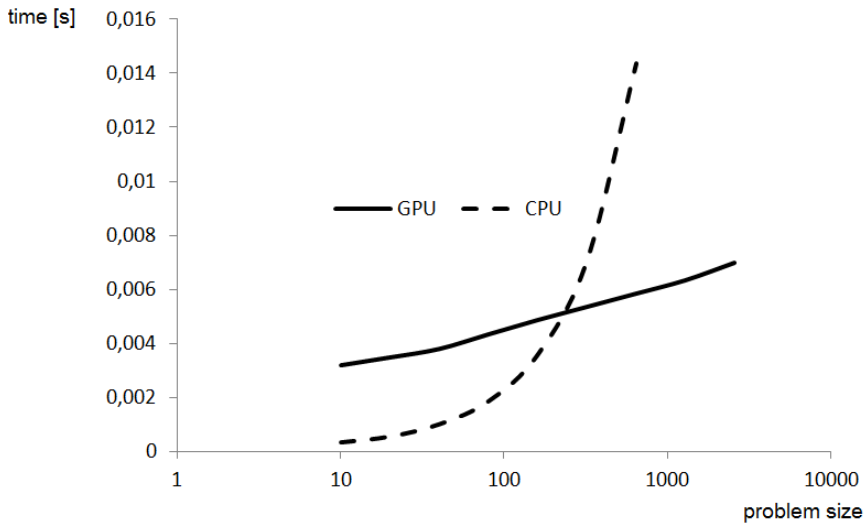

Figure 7: One dimensional solver using quartic B-splines. Comparison of the GPU cost vs the CPU one.

presented in Figure 8, since we are not able to modify the number of working cores in the GPUs we employ.

Based on the strong scalability results, we present in Figures 10 and 11 the speedup $S = T_1/T_p$ and efficiency $E = T_1/(p * T_p)$ of the GPU solver.

The matrices from the leaves of the elimination tree are generated in the global memory of the GPU, and transferred to the GPU computing nodes in order to be partially factorized. All partially factorized matrices are transferred back to global memory and must be kept there until the backward substitution step. The amount of available global memory does not influence the scalability of the solver algorithm. It just influences the size of the problem that can be solved on GPU. Once the problem cannot fit into GPU memory, we cannot solve it anymore.

7.2. 2D case

The 2D simulations have been performed over two graphic devices, namely NVIDIA Tesla C2070, and NVIDIA GTX 780 graphic card.

We start with NVIDIA Tesla C2070 experiments, presented in Figures 12-14, for linear, quadratic and cubic B-splines. The red line entitled "GPU" corresponds to the 2D GPU solver execution time, the green line entitled "GPU (estimate)" corresponds to the theoretical estimate for the parallel

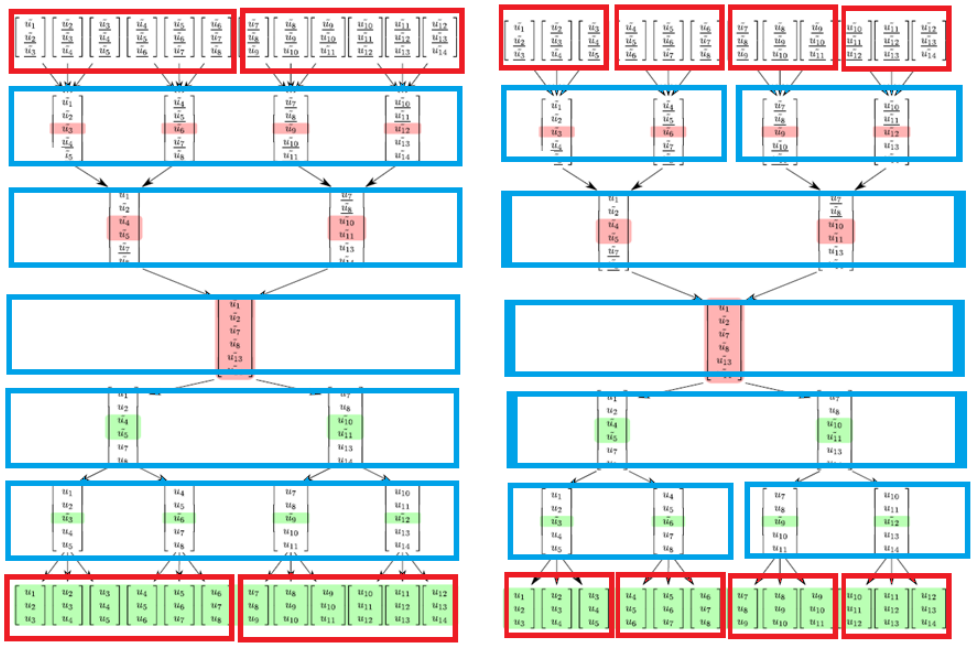

Figure 8: Comparison on the effect of decreasing the number of processors on the number of sets of data that can be processed fully in parallel, set by set.

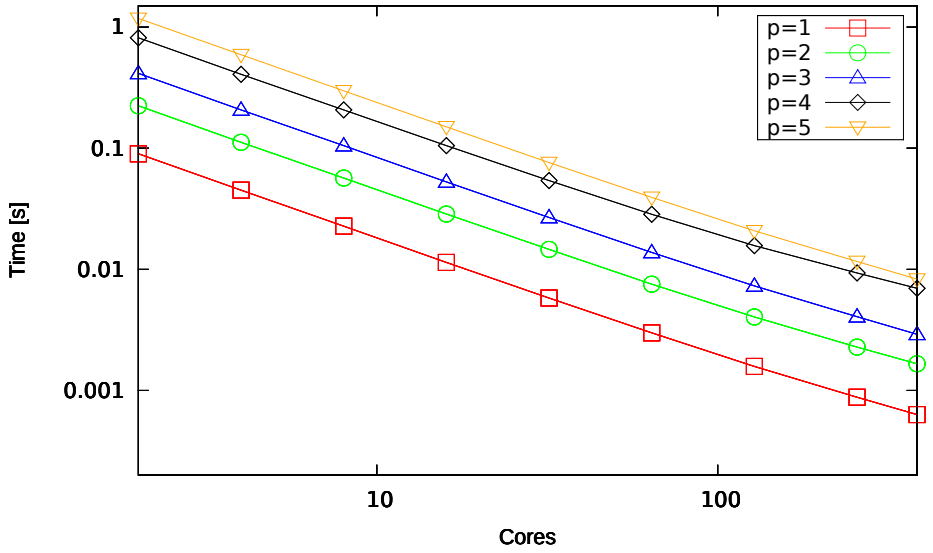

Figure 9: Strong scalability of the one-dimensional GPU solver for 2048 elements using linear, quadratic, cubic, and quartic B-splines. The plot was obtained by theoretical analysis.

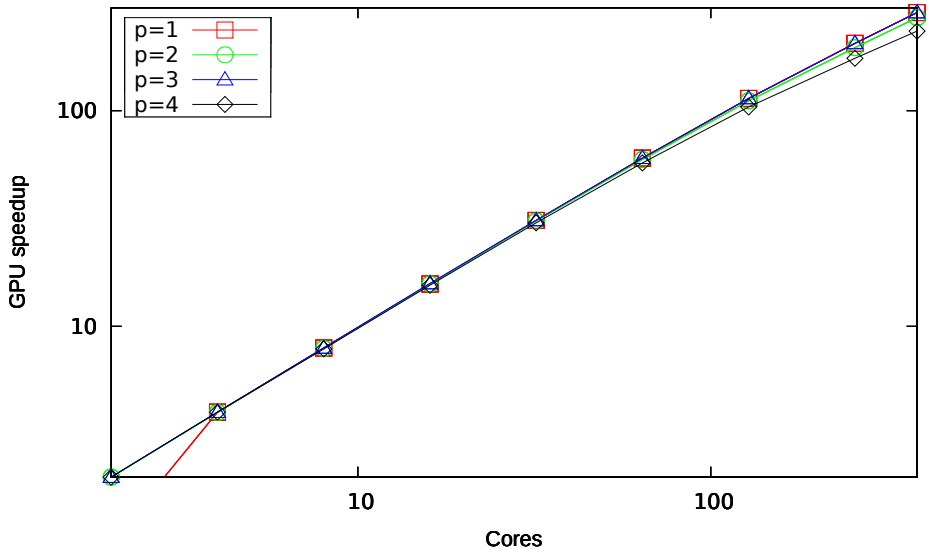

Figure 10: The speedup of the 1D GPU solver for 2048 elements, for linear, quadratic, cubic, and quartic B-splines. The plot was obtained by theoretical analysis.

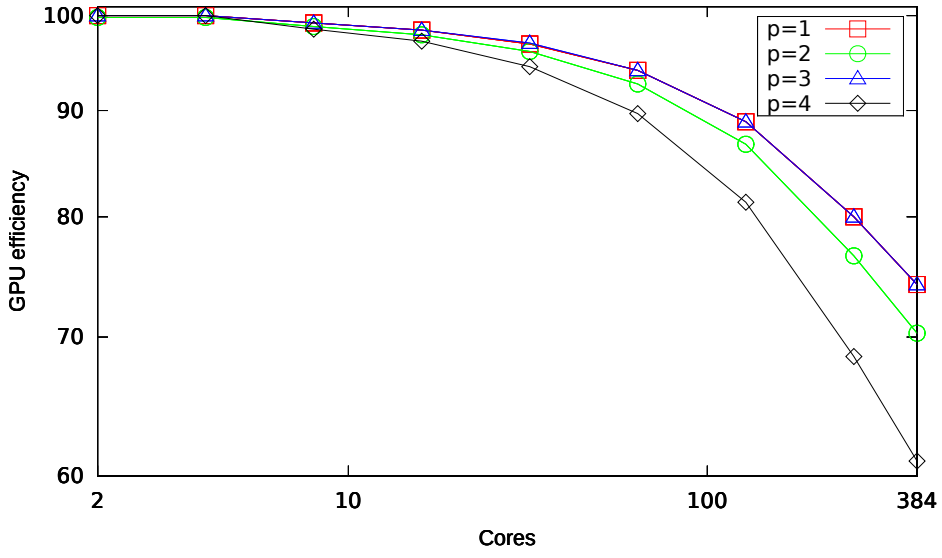

Figure 11: The efficiency of the 1D GPU solver for 2048 elements, for linear, quadratic, cubic, and quartic B-splines. The plot was obtained by theoretical analysis.

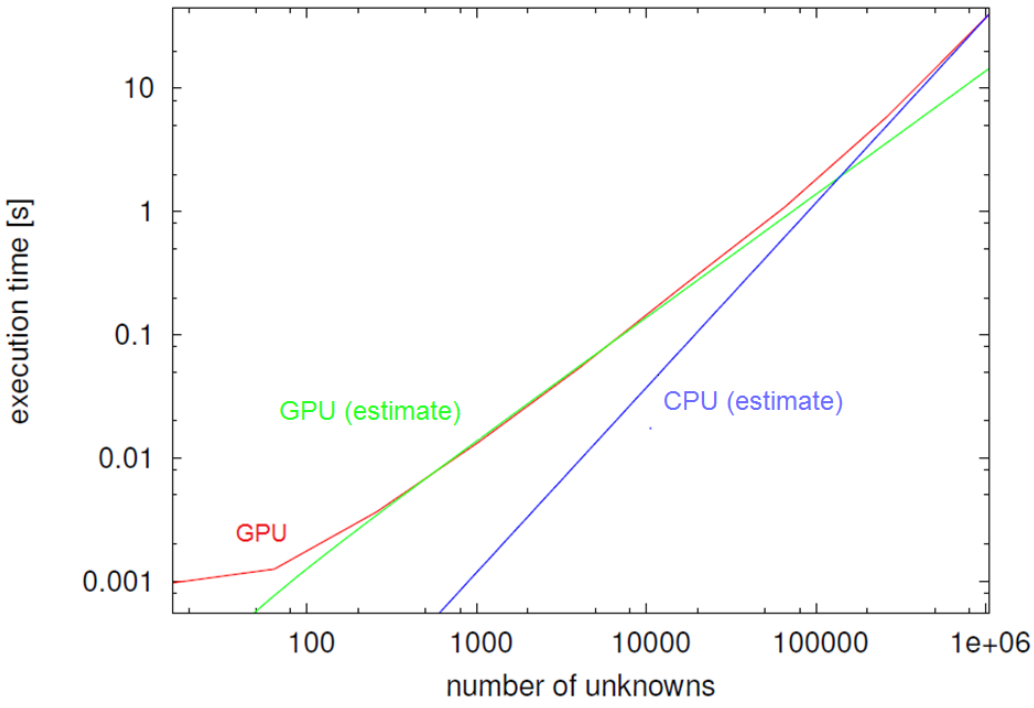

Figure 12: Scalability of the two-dimensional GPU solver compared with the theoretical cost for: (a) an ideal shared memory solver GPU (estimate) and (b) a sequential solver CPU (estimate), for linear B-splines.

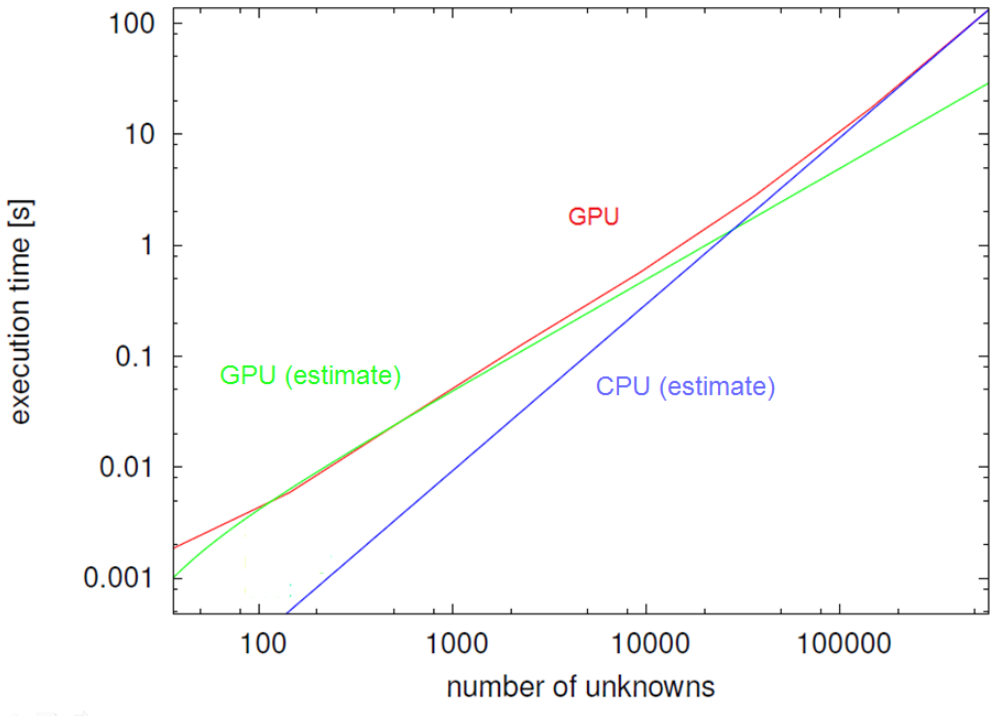

Figure 13: Scalability of the two-dimensional GPU solver compared with the theoretical cost for: (a) an ideal shared memory solver GPU (estimate) and (b) a sequential solver CPU (estimate), for quadratic B-splines.

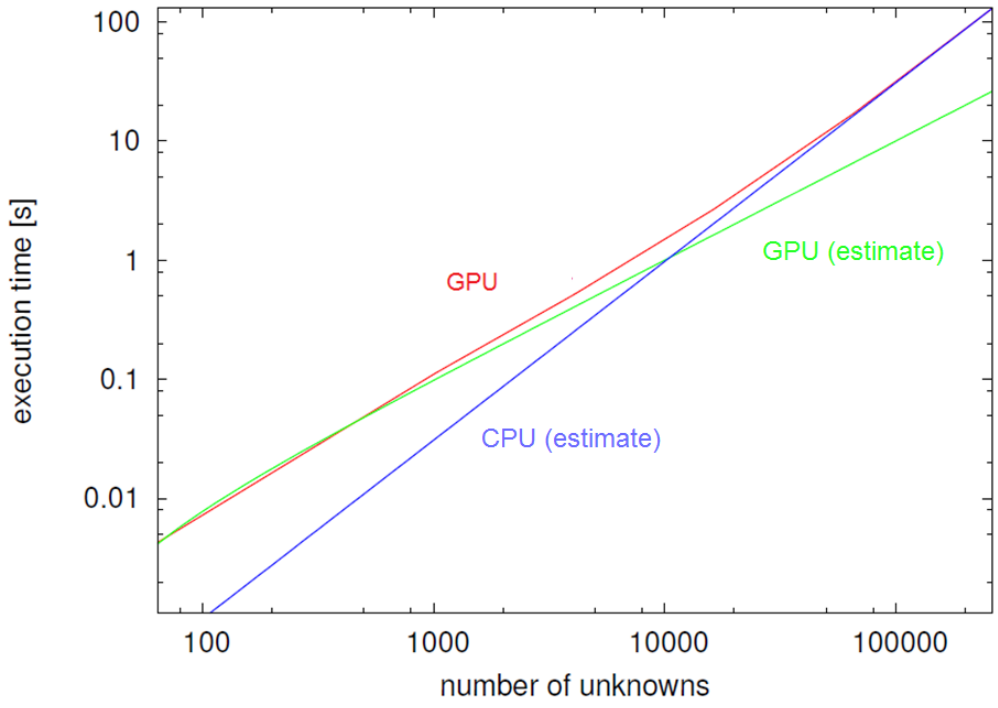

Figure 14: Scalability of the two-dimensional GPU solver compared with the theoretical cost for: (a) an ideal shared memory solver GPU (estimate) and (b) a sequential solver CPU (estimate), for cubic B-splines.

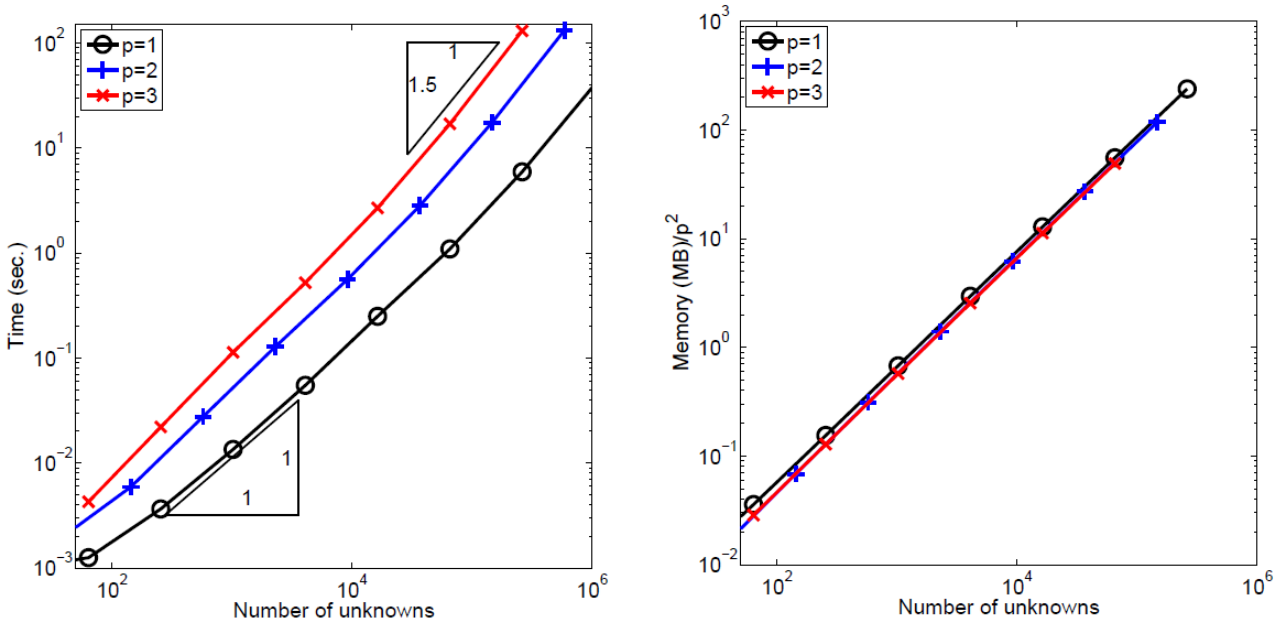

Figure 15: Execution time (left) and memory usage (right) measured on GPU for linear, quadratic, and cubic 2D B-splines solver.

solver, the blue line entitled "CPU (estimate)" corresponds to the sequential solver theoretical cost. The 2D GPU solver is supposed to deliver $O(Np^2)$ computational cost, which for fixed p results in $O(N)$ behavior. This is a small GPU with small number of cores, so the GPU solver behaves like parallel solver for small problem sizes, but it converges to the sequential solver cost for large problems. When the number of cores is too low with respect to the problem size, the GPU solver approaches the sequential cost, and delivers $O(N^{1.5}p^2)$ computational cost. Both $O(N)$ and $O(N^{1.5})$ costs become straight lines on the log-log scale, but with different slope. We have fitted the $O(N)$ straight line —GPU(estimate)— to the GPU plot, as well as the $O(N^{1.5})$ straight line —CPU(estimate)— to the CPU plot.

In Figure 15 we report the time spent on execution of the multi-frontal solver, as well as the memory usage. The numerical experiments were performed with linear, quadratic, and cubic B-splines. In Fig. 15 (left panel), we observe straight lines of slope 1 in the log-log scale in the pre-asymptotic regime, as predicted by the theory. For a number of unknowns significantly

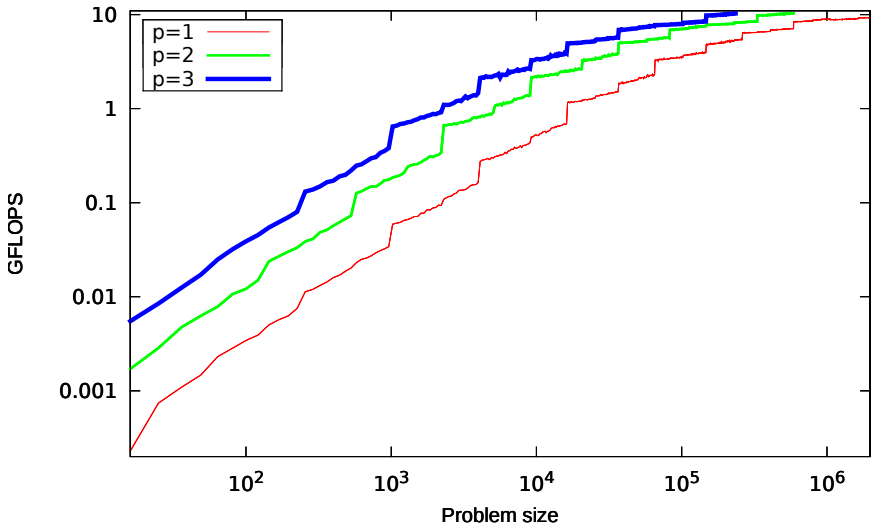

Figure 16: Measurements of GFLOPS on NVIDIA GTX 780 for linear, quadratic and cubic B-splines.

larger than the number of available cores, this slope becomes 1.5, which corresponds to the sequential estimates. In terms of memory estimates – Fig.15 right panel – we obtain straight lines of slope 1 in the log-log scale. Since the memory has been divided by p^2 according to the theoretical estimates, we obtain that all curves coincide, as expected.

Additionally, we have estimated the number of GFLOPS and memory usage of our GPU solver. We measured both GFLOPS and memory on NVIDIA GTX 780 for all possible sizes, to capture local variations. GFLOPS are estimated by measuring floating point operations by `nvprof` command, which returns average operations count in kernel as well as total kernels count. The GFLOPS measurements are presented in Figure 16. The GFLOS are estimated by counting the number of total floating point operations on GPU divided by the total execution time. The reference GFLOPS capability of NVIDIA GTX 780 is equal to 166 GFLOPS for double precision.

Our solver delivers 10 GFLOPS (which is 6 percent of the theoretical peack performance). Notice that our numerical results are intended to illustrate the predicted scalability results, but it is not our aim to show that our implementation is optimal.

Memory complexity is estimated by running algorithm and testing RAM

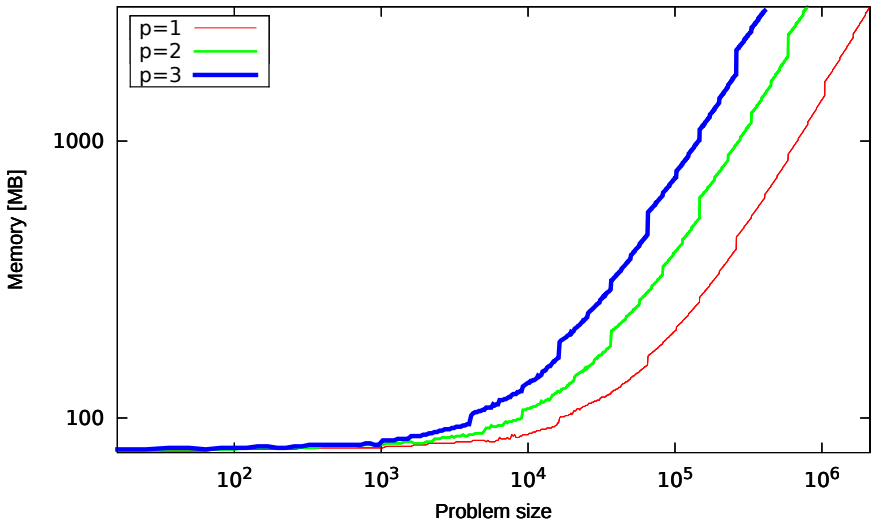

Figure 17: Measurements of memory usage on NVIDIA GTX 780 for linear, quadratic and cubic B-splines. Logarithmic scale is utilized on both horizontal and vertical axis.

status on GPU every 1ms. We used `nvidia-smi` command to test memory usage on GPU. The memory usage measurements are presented in Figure 17.

We have also compared our solver over GeForce GTX 780 device with the sequential MUMPS solver as well as with theoretical estimates for the ideal shared memory machine. The sequential MUMPS solver was executed over AMD Opteron 1220, 2.80GHz, compiled with intel fortran compiler, utilizing a single core. The 2D results are presented in Figures 21-23, where we change the problem size and global polynomial order of approximation p=1,2,3, resulting in C^0, C^1 and C^2 global continuity.

We start presenting the execution times for all parts of the CPU algorithm. This includes integration, factorization, and backward substitution time. The results are presented in Figures 18-20, for first, second and third order B-splines. In particular, we also distinguish the time spent by our solver on MAGMA dense solver calls [39] used to solve the root problem from our elimination tree. The MAGMA solver is also of the same (or lower) cost than the factorization cost, however the constant is quite large, and this is why for small problem sizes our solver execution time is limited by the MAGMA call time. The dominating term for all the cases is the factorization, in other

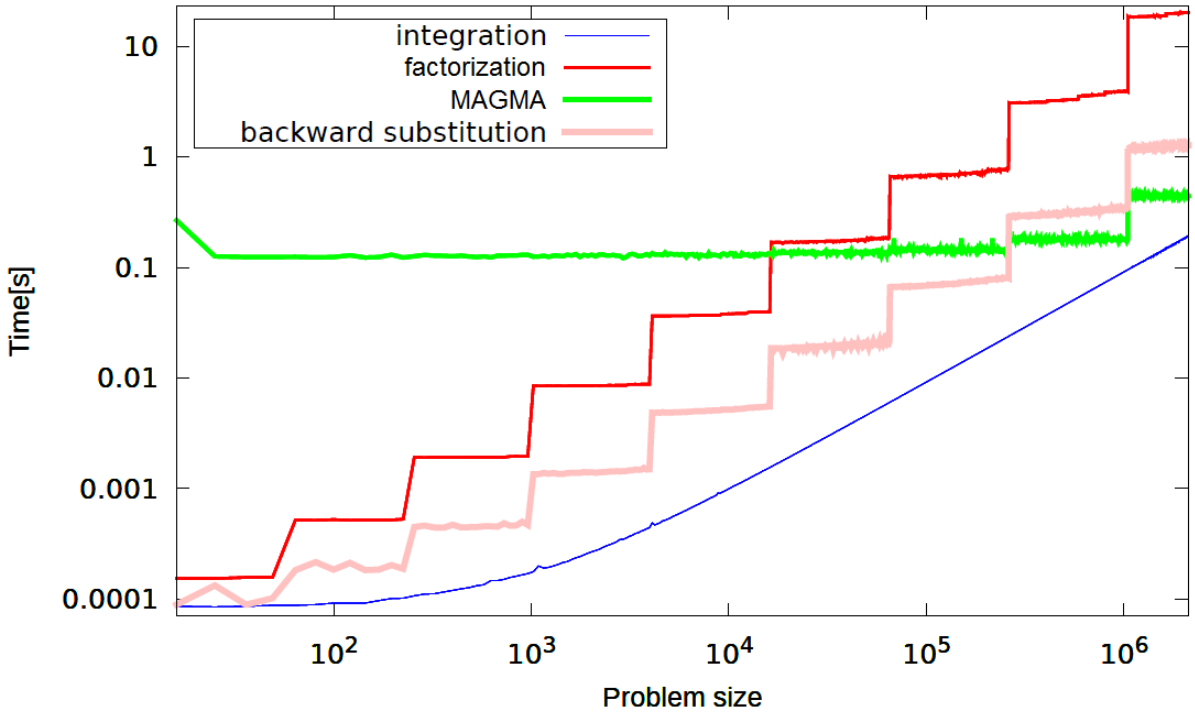

Figure 18: Measurements of the execution times for all parts of the GPU solver for linear B-splines, presented on a log-log scale .

words, the integration and backward substitution execution times are of the same or smaller order than the factorization time, thus, they do not influence the trends (scalability) of the computational cost, they only affect the constants.

In Figures 21-27 we focus on the total execution times. The solver can deal with increasing problem size until the memory of the GPU card is full. For linear B-splines, the solver can solve up to $2,128,681$ degrees of freedom with C^0 global continuity, for quadratic B-splines up to $797,449$ degrees of freedom with C^1 continuity, for cubic B-splines up to $408,321$ degrees of freedom with C^2 global continuity.

We start with presenting the comparison of the total GPU execution time versus total CPU MUMPS solver execution time, as presented in Figures 21-23. Next, in Figures 24-26 we execute a curve fitting algorithm to estimate the exponent factors of the lines plot on the log-log scale. To do that, we

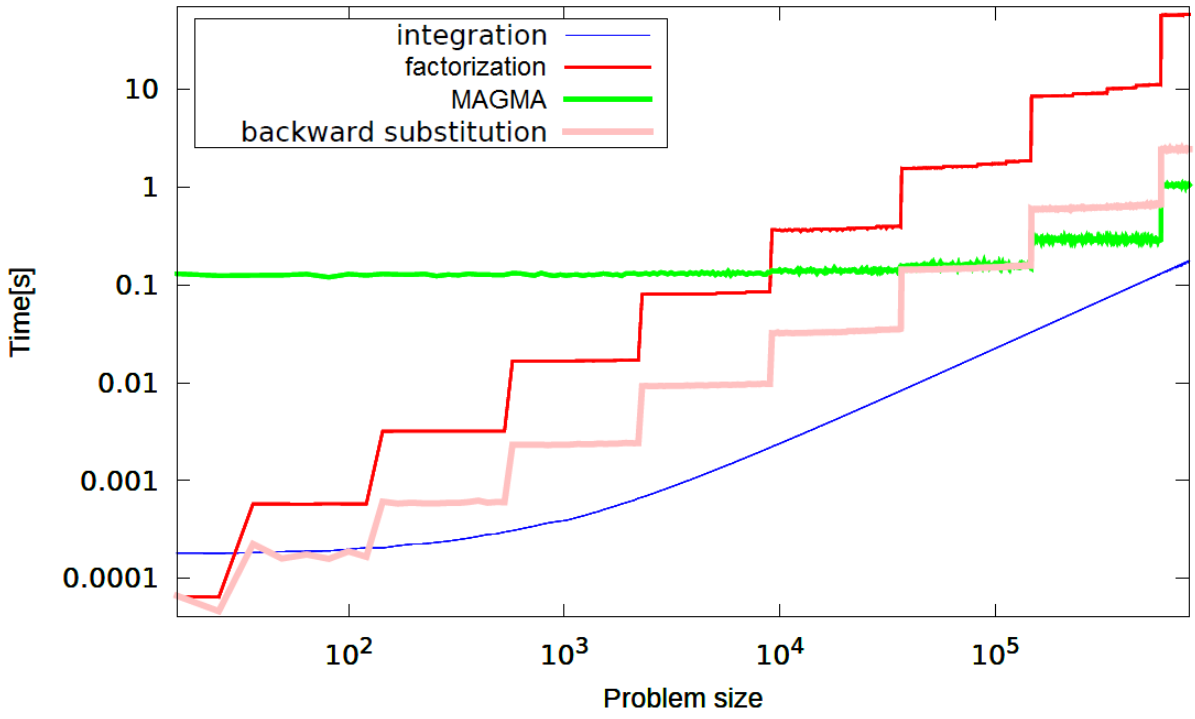

Figure 19: Measurements of the execution times for all parts of the GPU solver for quadratic B-splines, presented on a log-log scale .

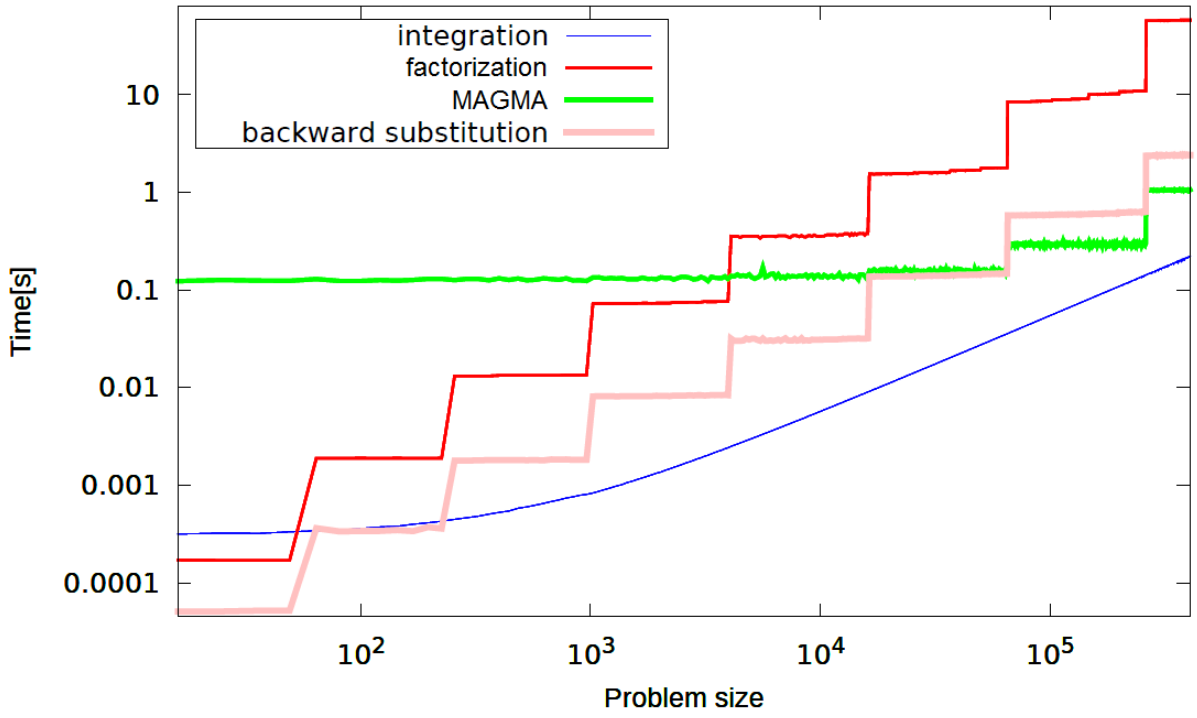

Figure 20: Measurements of the execution times for all parts of the GPU solver for cubic B-splines, presented on a log-log scale .

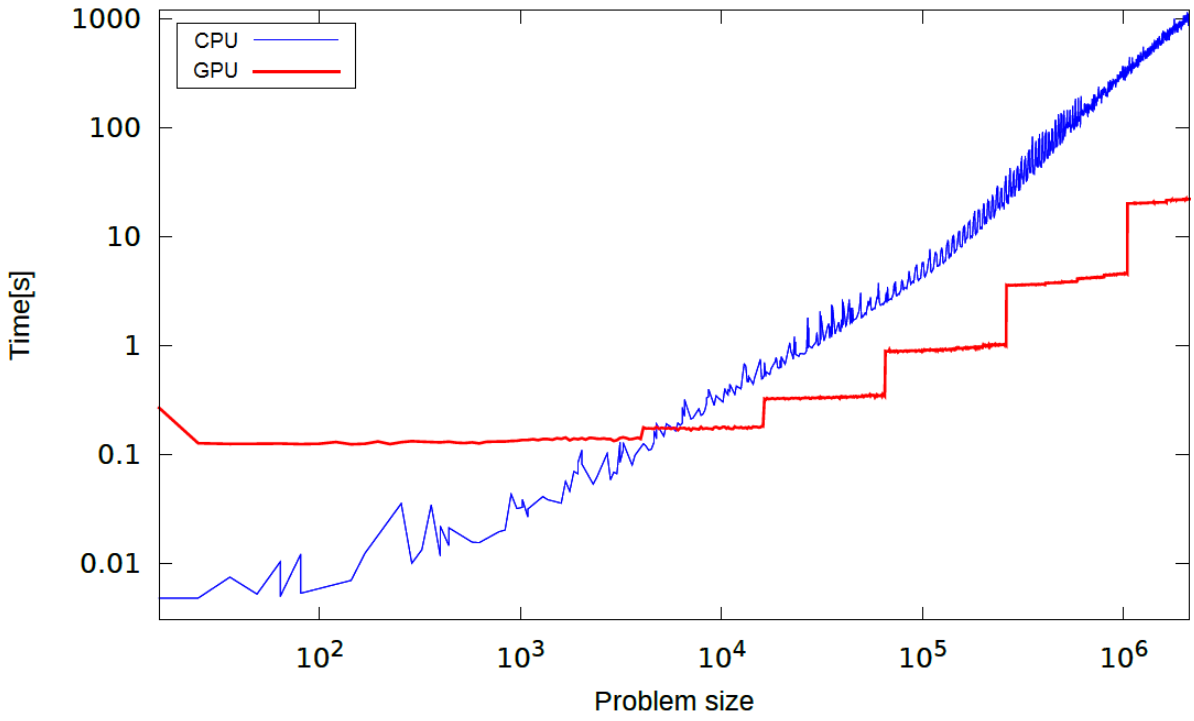

Figure 21: Comparison of total execution time of GPU solver versus MUMPS sequential CPU solver for linear B-splines, presented on a log-log scale. The maximum problem size for GPU solver is 2.1 million degrees of freedom with C^0 global continuity.

skip the first part of the plots, where for the GPU solver the MAGMA call dominates the cost, compare Figures 18-20.

The GPU solver is supposed to deliver $O(Np^2)$ computational cost, which for fixed p results in $O(N)$ behavior. The CPU solver is supposed to deliver $O(N^{1.5}p^2)$ computational cost, which for fixed p becomes $O(N^{1.5})$ behavior. Both $O(N)$ and $O(N^{1.5})$ costs become straight lines on the log-log scale, but with different slope. Actually, we have executed the curve fitting algorithm and we have obtained the following exponents: for CPU results, 1.3504 for $p = 1$; 1.478 for $p = 2$, and 1.471 for $p = 3$. For GPU results 0.9515 for $p = 1$; 1.0746 for $p = 2$, and 1.0490 for $p = 3$. These results confirm our expected scalability of the solver. Since the employed GPU is large, it has a large number of cores, so even for rather large problems as the ones considered here, the GPU solver behaves like a parallel solver (it does not converge

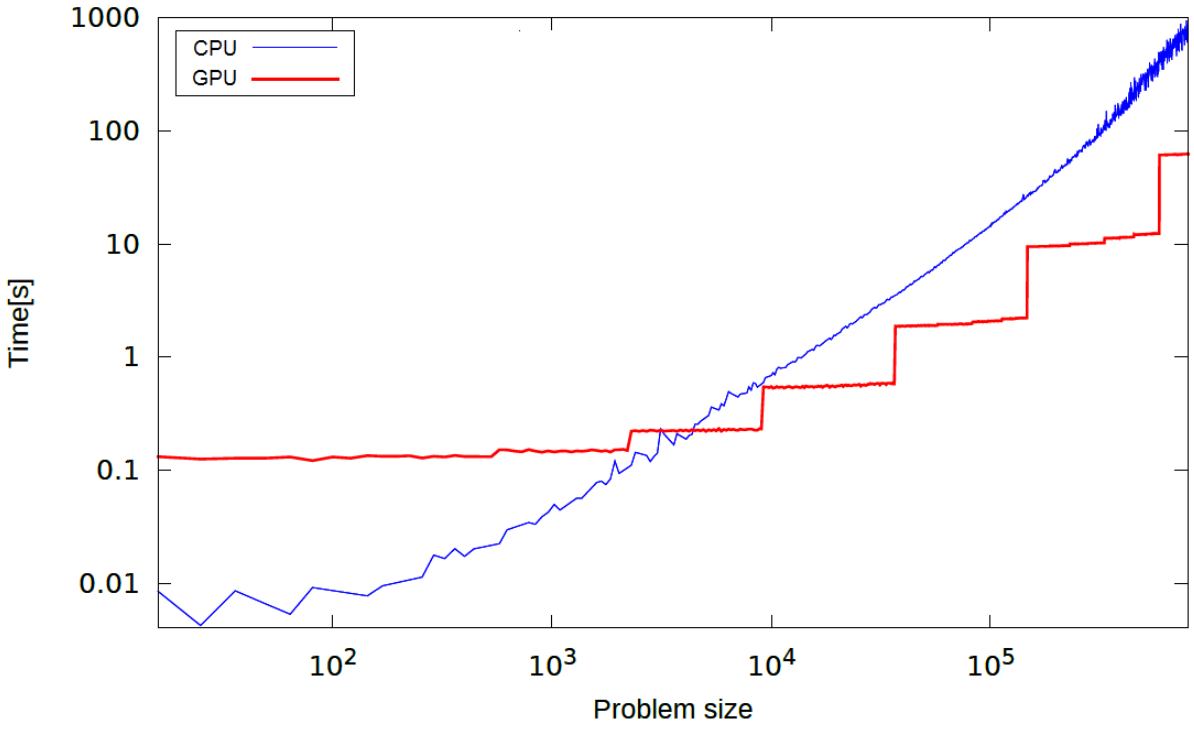

Figure 22: Comparison of total execution time of GPU solver versus MUMPS sequential CPU solver for quadratic B-splines, presented on a log-log scale. The maximum problem size for GPU solver is 0.8 million degrees of freedom with C^1 global continuity.

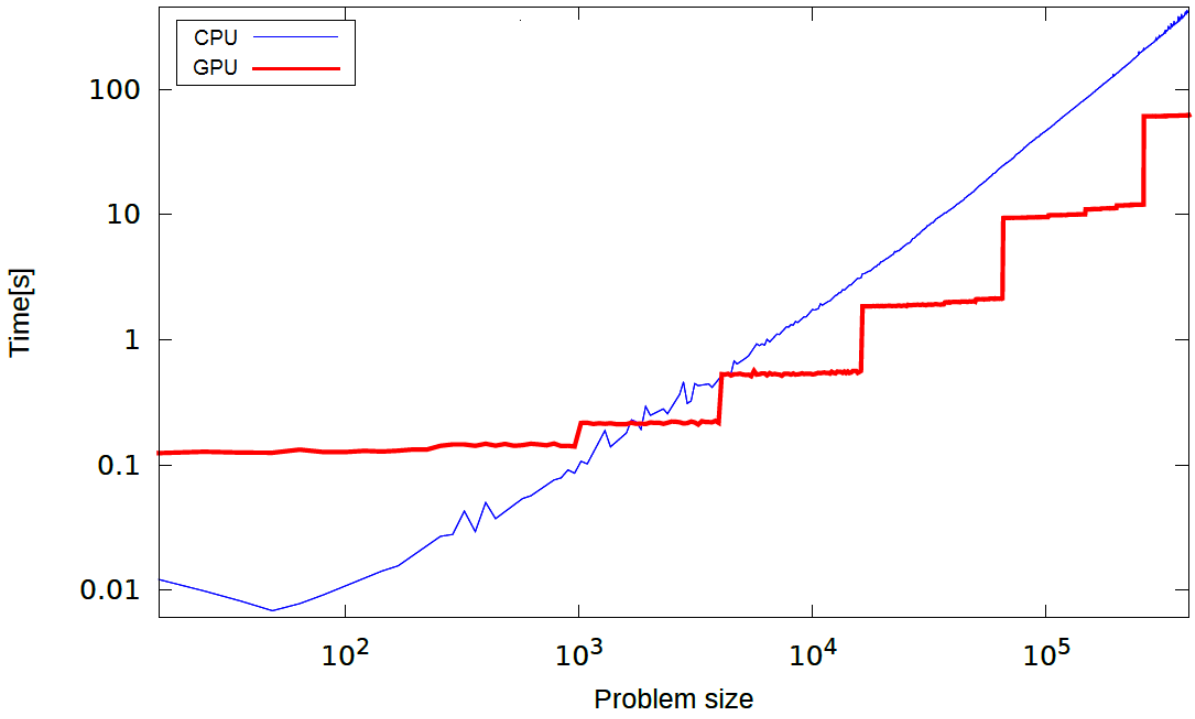

Figure 23: Comparison of total execution time of GPU solver versus MUMPS sequential CPU solver for cubic B-splines, presented on a log-log scale. The maximum problem size is 0.4 million degrees of freedom with C^2 global continuity.

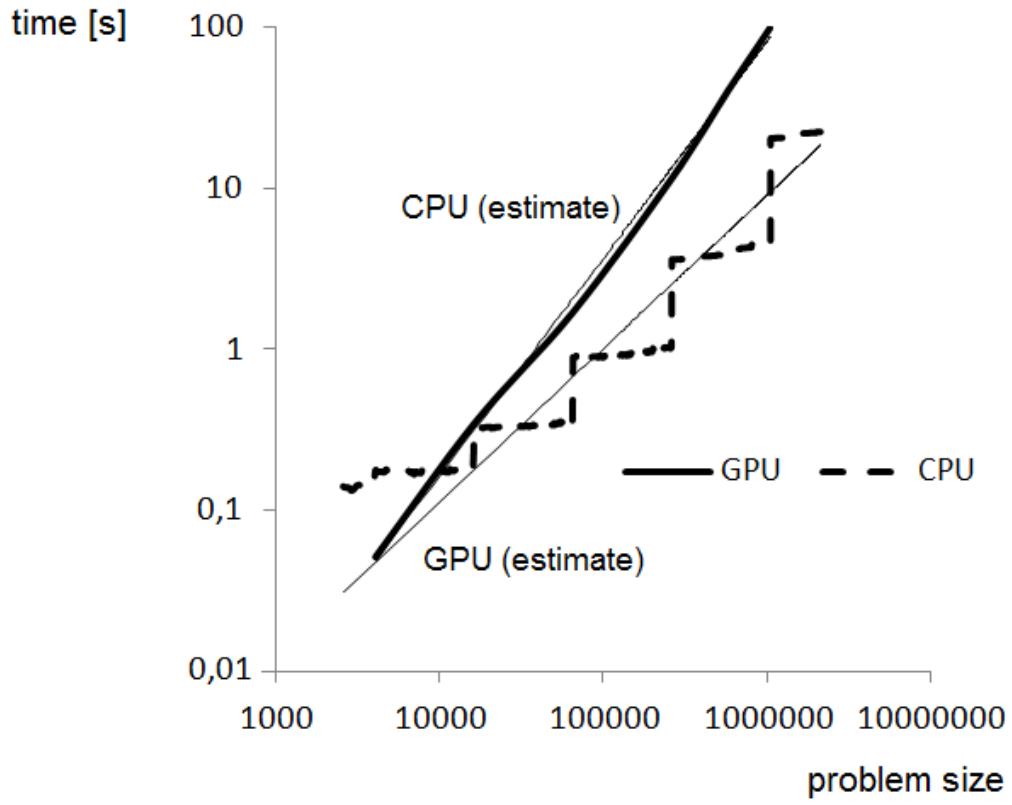

Figure 24: Scalability of the two-dimensional GPU solver on GeForce GTX 780 device compared with the scalability of the sequential MUMPS solver, for linear B-splines.

towards the sequential cost). Additionally, if we divide the GPU execution times by p^2, the numerical result curves coincide for $p = 1, 2, 3$, which proves the agreement with the theoretical result. This is illustrated in Figure 27.

7.3. Three dimensional (3D) case

The predicted scalability of 3D results in terms of memory prevent GPU's alone from being a good alternative to solve large problems discretized by IGA using direct solvers. The amount of memory to solve significant 3D problems is of the order $O\left(p^2 N^{4/3}\right)$. Actually, this is the size of the top dense problem. And the amount of memory required to store all the frontal matrices from other nodes of the elimination tree is of the same order. In

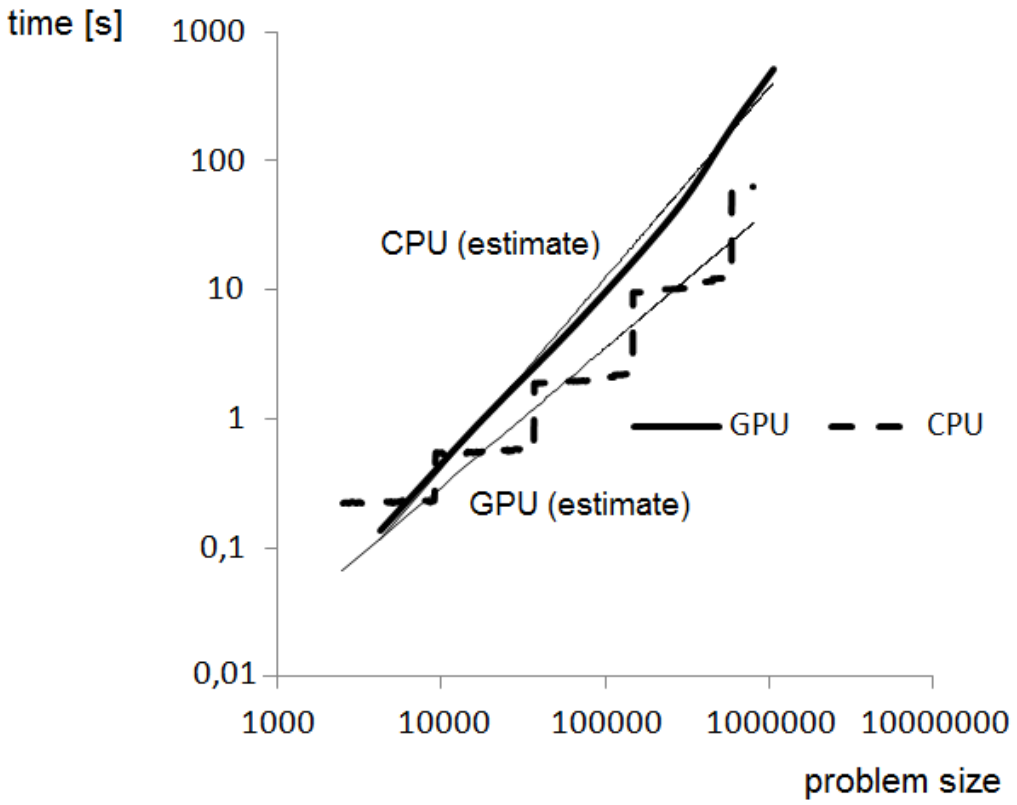

Figure 25: Scalability of the two-dimensional GPU solver on GeForce GTX 780 device compared with the scalability of the sequential MUMPS solver, for quadratic B-splines.

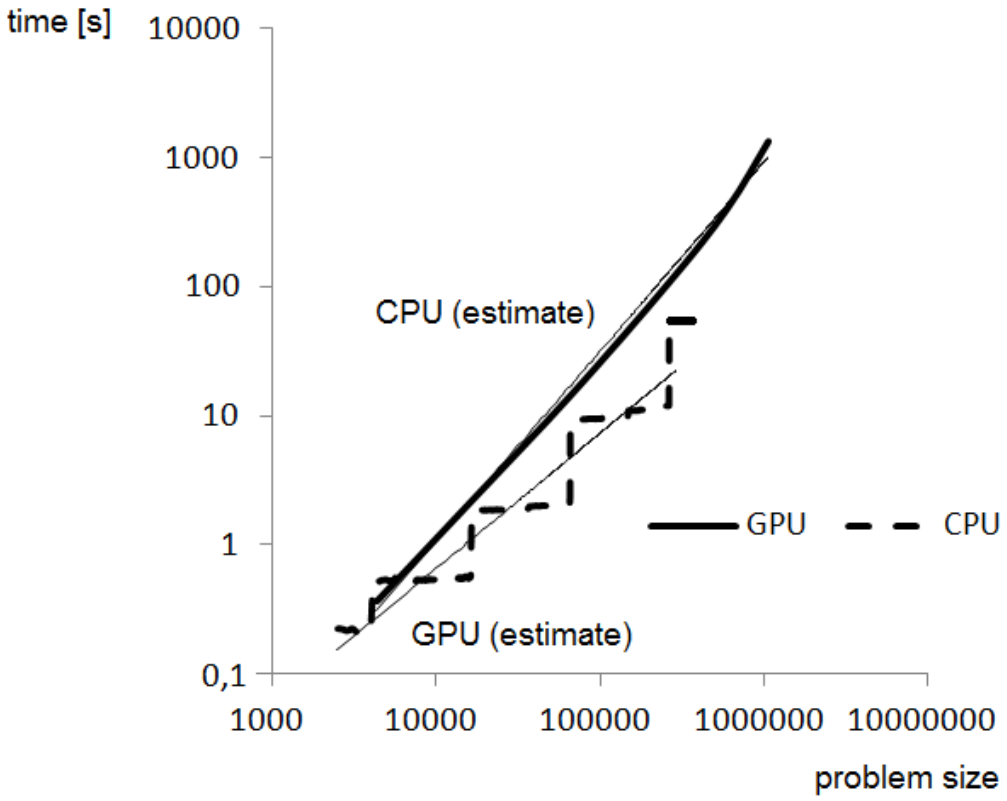

Figure 26: Scalability of the two-dimensional GPU solver on GeForce GTX 780 device compared with the scalability of the sequential MUMPS solver, for cubic B-splines.

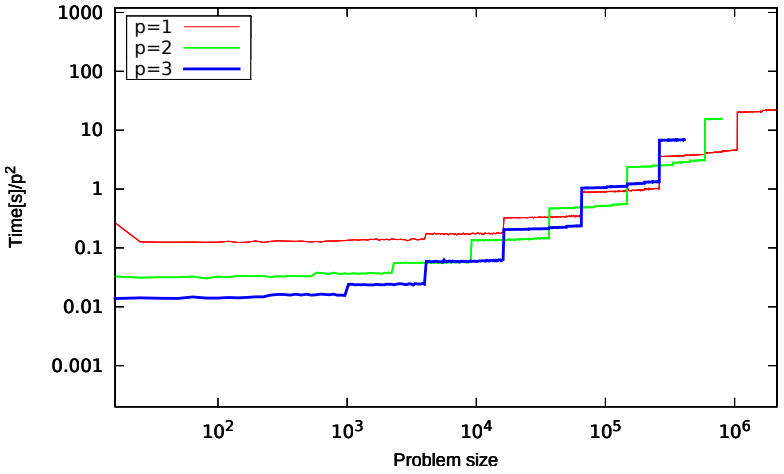

Figure 27: Execution times of the two-dimensional GPU solver on GeForce GTX 780 device for linear, quadratic and cubic B-splines, divided by p^2 factor.

other words, e.g. $500,000$ degrees of freedom for C^2 continuity with cubic B-splines requires storage of $8 \times 2 \times 3^2 \times 500,000^{4/3} = 5.5GB$, where 8 bytes are for double precision storage. Moreover, this estimate assumes ideal utilization of the GPU memory, and the amount of required memory is actually larger, since we need space for actual merging of frontal matrices. Thus, other authors employ hybrid CPU/GPU solvers [38], where the scalability estimates differ from those presented in this article.

Moreover, notice that in some 3D geophysical applications, the resulting system can be reduced into a system of 2D problems by employing a Fourier series expansion in a space dimension, as described in [42, 43]. Thus, the development of efficient 2D solvers is critical also from the point of view of 3D problems.

8. Conclusions and future work

In this paper we presented theoretical estimates of the computational cost and memory usage for parallel shared memory IGA solver. We showed that the ideal shared memory solver with infinite memory and zero communication cost delivers $O(p^2 \log(N/p))$ for 1D, $O(Np^2)$ for 2D, and $O(N^{4/3}p^2)$ for 3D problems, when using global $C^{(p-1)}$ continuity. We also showed that

the memory usage of such shared memory solver scales like $O(Np)$ in 1D, $O(p^2 N \log(N/p))$ in 2D, and $O(Np^2 N^{4/3})$ in 3D. We concluded that the shared memory IGA solver scales like p^2 when we increase the continuity of the solution. This is an important advantage with respect to sequential IGA solvers, which scale like p^3 when we increase the continuity, see [13]. The theoretical estimates presented in the paper have been verified by numerical experiments. In particular, we performed experiments on three GPU's, GeForce GTX 560Ti, NVIDIA Tesla C2070 and GeForce GTX 780. The obtained numerical results confirmed the theoretical estimates. We showed experimentally that the factorization cost dominates all other parts of the algorithm, and other factors as e.g. integration, memory access etc. are of the same or lower order, and thus, they only influence the constants in the computational cost estimate. The GPU implementation described in the paper forms an example implementation with the performance results being in accordance with theoretical estimates. A shared memory implementation for CPUs would also form a perfect example to illustrate the estimates derived in the paper. As future work, we consider to extend the results to T-splines, which allow for local adaptivity [19].

9. Acknowledgments

The work of KK was supported by the Polish National Science Center grant no. NN 519 447739. The work of MP was supported by the Polish National Science Center grants no. NN 519 447739 and DEC-2011/01/B/ST6/ 00674. The work of MW was supported by Polish National Science Grants no. DEC-2011/01/B/ST6/ 00674 and 2012/07/B/ST6/01229. The work of DP was partially funded by the Project of the Spanish Ministry of Sciences and Innovation MTM2010-16511, the Laboratory of Mathematics (UFI 11/52), and the IberoAmerican Project CYTED 2011 (P711RT0278).

References

[1] I. Akkerman, Y. Bazilevs, V. M. Calo, T. J. R. Hughes, S. Hulshoff, *The role of continuity in residual-based variational multiscale modeling of turbulence*, Computational Mechanics 41 (2008) 371–378.

[2] I. Babuska, B.A. Szabo, I.N. Katz, *The p-Version of the Finite Element Method*, SIAM Journal on Numerical Analysis, 18 (1981) 515-545.

[3] Y. Bazilevs, L. Beirao da Veiga, J.A. Cottrell, T.J.R. Hughes, and G. Sangalli, *Isogeometric analysis: Approximation, stability and error estimates for h-refined meshes*, Mathematical Methods and Models in Applied Sciences, 16 (2006) 1031–1090.

[4] Y. Bazilevs, V. M. Calo, J. A. Cottrell, T. J. R. Hughes, A. Reali, G. Scovazzi, *Variational multiscale residual-based turbulence modeling for large eddy simulation of incompressible flows*, Computer Methods in Applied Mechanics and Engineering 197 (2007) 173-201.

[5] Y. Bazilevs, V. M. Calo, Y. Zhang, T. J. R. Hughes, *Isogeometric fluid-structure interaction analysis with applications to arterial blood flow*, Computational Mechanics 38 (2006).

[6] D.J. Benson, Y. Bazilevs, E. De Luycker, M.C. Hsu, M. Scott, T.J.R. Hughes, and T. Belytschko, *A Generalized Element Formulation for Arbitrary Basis Functions: From Isogeometric Analysis to XFEM*, International Journal for Numerical Methods in Engineering 83 (2010) 765-785.

[7] D.J. Benson, Y. Bazilevs, M.-C. Hsu and T.J.R. Hughes, *A Large-Deformation, Rotation-Free Isogeometric Shell*, Computer Methods in Applied Mechanics and Engineering, 200 (2011) 1367-1378.

[8] C. Canuto, A. Quarteroni, M.Y. Hussaini, and T. A. Zang, *Spectral Methods: Fundamentals in Single Domains*, Springer-Verlag (2006).

[9] V.M. Calo, N. Brasher, Y. Bazilevs, and T.J.R. Hughes, "Multiphysics Model for Blood Flow and Drug Transport with Application to Patient-Specific Coronary Artery Flow," *Computational Mechanics*, 43(1) (2008) 161–177.

[10] V.M. Calo, N. O. Collier, D. Pardo, M. Paszyński, *Computational complexity and memory usage for multi-frontal direct solvers used in p finite element analysis,* Procedia Computer Science, 4 (2011) 1854-1861.

[11] K. Chang, T.J.R. Hughes, and V.M. Calo, "Isogeometric Variational Multiscale Large-Eddy Simulation of Fully-developed Turbulent Flow over a Wavy Wall," *Computers and Fluids*, 68 (2012) 94-104.

[12] N. Collier, L. Dalcin, V. M. Calo., "PetIGA: High-performance isogeometric analysis". arxiv, (1305.4452), (2013) http://arxiv.org/abs/1305.4452

[13] N.O. Collier, D. Pardo, M. Paszyński, L. Dalcín, and V. M. Calo, *The cost of continuity: a study of the performance of isogeometric finite elements using direct solvers,* Computer Methods in Applied Mechanics and Engineering, 213-216 (2012) 353-361.

[14] J.A. Cottrel, T.J.R. Hughes, Y. Bazilevs *Isogeometric Analysis. Toward Integration of CAD and FEA*, Wiley, (2009).

[15] L. Dedè,T. J. R. Hughes, S. Lipton, V. M. Calo, *Structural topology optimization with isogeometric analysis in a phase field approach*, USNC-TAM2010, 16th US National Congree of Theoretical and Applied Mechanics.

[16] L. Dedè, M. J. Borden, T. J. R. Hughes, *Isogeometric Analysis for topology optimization with a phase field model*, ICES REPORT 11-29, The Institute for Computational Engineering and Sciences, The University of Texas at Austin (2011).

[17] L. Demkowicz, *Computing with hp-Adaptive Finite Element Method. Vol. I.One and Two Dimensional Elliptic and Maxwell Problems*. Chapmann & Hall / CRC Applied Mathematics & Nonlinear Science (2006).

[18] L. Demkowicz, J. Kurtz, D. Pardo, M. Paszyński, W. Rachowicz, A. Zdunek, *Computing with hp-Adaptive Finite Element Method. Vol. II. Frontiers: Three Dimensional Elliptic and Maxwell Problems*. Chapmann & Hall / CRC Applied Mathematics & Nonlinear Science (2007).

[19] M. R. Dorfel, B. Juttler, B. Simeon, *Adaptive isogeometric analysis by local h-refinement with T-splines*, Computer Methods in Applied Mechanics and Engineering, 199, 58 (2010) 264–275.

[20] R. Duddu, L. Lavier, T.J.R. Hughes, and V.M. Calo. "A finite strain Eulerian formulation for compressible and nearly incompressible hyperelasticity using high-order NURBS elements," *International Journal of Numerical Methods in Engineering*, 89(6) (2012) 762-785.

[21] I. S. Duff, J. K. Reid, *The multifrontal solution of indefinite sparse symmetric linear systems*. ACM Transactions on Mathematical Software. 9 (1983) 302–325.

[22] I. S. Duff, J. K. Reid *The multifrontal solution of unsymmetric sets of linear systems*, SIAM Journal on Scientific and Statistical Computing, 5 (1984) 633–641.

[23] S. Fialko, *A block sparse shared-memory multifrontal finite element solver for problems of structural mechanics*. Computer Assisted Mechanics and Engineering Sciences, 16 (2009) 117-131.

[24] S. Fialko, *The block subtracture multifrontal method for solution of large finite element equation sets*. Technical Transactions, 1-NP, 8 (2009) 175–188.

[25] S. Fialko, *PARFES: A method for solving finite element linear equations on multi-core computers*. Advances in Engineering Software, 40(12) (2010) 1256–1265.

[26] P. Geng, T. J. Oden, R. A. van de Geijn; *A Parallel Multifrontal Algorithm and Its Implementation*, Computer Methods in Applied Mechanics and Engineering, 149 (2006) 289–301.

[27] L. Giraud, A. Marocco, J.-C. Rioual, *Iterative versus direct parallel substructuring methods in semiconductor device modeling*. Numerical Linear Algebra with Applications, 12(1) (2005) 33–55.

[28] H. Gómez, V. M. Calo, Y. Bazilevs, T. J. R. Hughes, *Isogeometric analysis of the Cahn-Hilliard phase-field model*, Computer Methods in Applied Mechanics and Engineering 197 (2008) 4333–4352.

[29] H. Gómez, T. J. R. Hughes, X. Nogueira, V. M. Calo, *Isogeometric analysis of the isothermal Navier-Stokes-Korteweg equations*, Computer Methods in Applied Mechanics and Engineering 199 (2010) 1828-1840.

[30] S. Hossain, S.F.A. Hossainy, Y. Bazilevs, V.M. Calo, and T.J.R. Hughes, "Mathematical Modeling of Coupled Drug and Drug-Encapsulated Nanoparticle Transport in Patient-Specific Coronary Artery Walls," *Computational Mechanics*, doi: 10.1007/s00466-011-0633-2, (2011).

[31] M.-C. Hsu, I. Akkerman, and Y. Bazilevs, High-performance computing of wind turbine aerodynamics using isogeometric analysis, Computers and Fluids, 49(1) (2011) 93-100.

[32] T.J.R. Hughes, A. Reali, G. Sangalli, Efficient quadrature for NURBS-based isogeometric analysis, Computer Methods in Applied Mechanics and Engineering, 199(5-8) (2010), 301-313.

[33] B. Irons, *A frontal solution program for finite-element analysis*. International Journal of Numerical Methods in Engineering, 2 (1970) 5–32.

[34] S. Hong, H. Kim. "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," Technical Report TR-2009-003, Atlanta, GA, USA, (2009).

[35] J. Sim, A. Dasgupta, H. Kim, R. Vuduc, "A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications," Proceedings of the 17th Symposium on Principles and Practice of Parallel Programming (PPoPP), New Orleans, LA (2012).

[36] K. Kuznik, M. Paszyński, V. Calo, *Graph Grammar-Based Multi-Frontal Parallel Direct Solver for Two-Dimensional Isogeometric Analysis*. Procedia Computer Science 9 (2012) 1454-1463.

[37] K. Kuznik, M. Paszyński, V. Calo, *Grammar-Based Multi-Frontal Solver for One Dimensional Isogemetric Analysis with Multiple Right Hand Sides* Procedia Computer Science 18 (2013) 1574-1583.

[38] X. Lacoste, P. Ramet, M. Faverge, Y. Ichitaro, J. Dongarra, Sparse direct solvers with accelerators over DAG runtimes, Technical Report N RR-7972 HAL-Inria (2012).

[39] MAGMA Matrix Algebra on GPU and Multicore Architecture, http://icl.cs.utk.edu/magma/

[40] MUlti-frontal Massivelly Parallel sparse direct Solver, http://graal.ens-lyon.fr/MUMPS/

[41] P. Obrok, P. Pierzchala, A. Szymczak, M. Paszyński *Graph grammar-based multi-thread multi-frontal parallel solver with trace theory-based scheduler*, Proceedia Computer Science, 1(1) (2010) 1993–2001.

[42] D. Pardo, V.M. Calo, C. Torres-Verdn, M. J. Nam *Fourier series expansion in a non-orthogonal system of coordinates for simulation of 3D DC borehole resistivity measurements*, Computer Methods in Applied Mechanics and Engineering, 197(1-3) (2008)1906-1925.

[43] D. Pardo, C. Torres-Verdn, M.J. Nam, M. Paszynski, V. M. Calo. *Fourier series expansion in a non-orthogonal system of coordinates for simulation of 3D alternating current borehole resistivity measurements* Computer Methods in Applied Mechanics and Engineering, 197 (2008) 3836-3849.

[44] M. Paszyński, D. Pardo, C. Torres-Verdin, L. Demkowicz, V. Calo *A Parallel Direct Solver for Self-Adaptive hp Finite Element Method*. Journal of Parall and Distributed Computing 70 (2010) 270–281.

[45] M. Paszyński, D. Pardo, A. Paszyńska, *Parallel multi-frontal solver for p adaptive finite element modeling of multi-physics computational problems*. Journal of Computational Science 1 (2010) 48–54.

[46] M. Paszyński, R. Schaefer *Graph grammar driven partial differential eqautions solver*. Concurrency and Computations: Practise and Experience 22(9) (2010) 1063–1097.

[47] J. A. Scott, Parallel Frontal Solvers for Large Sparse Linear Systems, ACM Trans. on Math. Soft., 29(4) (2003) 395-417.

[48] B. F. Smith, P. Bjørstad, W. Gropp, *Domain Decomposition, Parallel Multi-Level Methods for Elliptic Partial Differential Equations*, Cambridge University Press, New York, 1st ed. (1996).

[49] Stothers, A. J. On the complexity of matrix multiplication, PhD. Thesis, The University of Edinburgh (2010) .

[50] A. Szymczak, M. Paszyński *Graph grammar based Petri net controlled direct sovler algorithm*. Computer Science 11 (2010) 65–79.

[51] Verhoosel, C. V., Scott, M. A., Hughes, T. J. R. and de Borst, R. (2011), An isogeometric analysis approach to gradient damage models, International Journal for Numerical Methods in Engineering, 86 (2011) 115134.